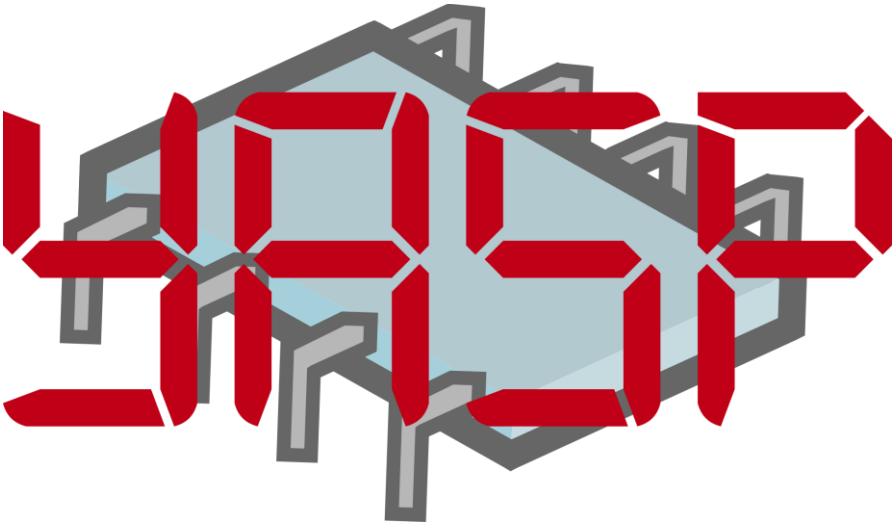
SPENGERGASSE
ausbildung mit zukunft

HTBLuVA Wien V
Höhere Lehranstalt für Informatik
Ausbildungsschwerpunkte Intelligente Systeme,
Tools und Methodik

htl
bildung mit zukunft



Abschlussdokumentation

Ausgeführt im Schuljahr 2013/14 von

Michael Lutonsky 5EHIF
Robert Fischer 5AHIF

Betreuer:

Dipl.- Ing. Robert Jelinek
Dipl.- Ing. Joachim Grüneis

Wien, am 14.04.2014

Eidesstattliche Erklärung

Ich erkläre an Eids statt, dass wir die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegeben Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Wien, am 14.04.2014

Michael Lutonsky, 5EHIF _____

Robert Fischer, 5AHIF _____

htl bildung mit zukunft	**HTBLuVA Wien V**	SPENGERGASSE ausbildung mit zukunft
	Abteilung:	Informatik
	Ausbildungsschwerpunkt:	Intelligente Systeme, Tools und Methodik

DIPLOMARBEIT
Dokumentation

Name der Verfasser/innen	Michael Lutonsky 5EHIF Robert Fischer 5AHIF
Jahrgang / Klasse Schuljahr	5AHIF, 2013/14 5EHIF, 2013/14
Thema der Diplomarbeit	Reimplementierung des USB-Masters im Web (samt Debugger, Assembler, Editor und Emulator)
Kooperationspartner	Höhere Technische Bundes-Lehr und Versuchsanstalt Spengergasse 20 A-1050

Aufgabenstellung	Das Ziel unseres Projekts ist die Bereitstellung einer plattformunabhängigen Entwicklungsumgebung, um Schülern und Schülerinnen eine komfortable Möglichkeit zu bieten, den Umgang mit Assembler zu erlernen. An der HTBLuVA Spengergasse wird dies derzeit mit einer eigens entwickelten Hardware realisiert. Unser Projekt wird diese ablösen, was bedeutet, dass die Schülerinnen und Schüler sich nicht mehr um das komplizierte Auf- und Abbauen der Hardware kümmern müssen. Sie können so eine zuverlässige Plattform verwenden und dadurch mehr Spaß beim Entwickeln haben. Weiters können sie auch zu Hause den Umgang mit Hard- und Softwareelementen lernen. Zusätzlich erleichtert es den Unterricht ungemein, da sich niemand Sorgen mehr um das Funktionieren der Hardware machen muss.

Realisierung	HTML5 (JavaScript, HTML, CSS), PHP

Ergebnisse	*yasp* ist unter http://demo.yasp.me/ erreichbar.

Abstract (Deutsch)	Das Ziel von yasp ist es das Lernen einer Assembler-Programmiersprache so einfach wie möglich zu gestalten. Die HTBLuVA Spengergasse setzt eine, ebenfalls an dieser Schule entwickelte Hardware ein. Mit dieser können die Schülerinnen und Schüler in einer vereinfachten Variante der Assembler-Programmiersprache programmieren und so den Umgang mit der Hardware erlernen. Da die diese händisch gelötet wird, ist sie leider sehr fehleranfällig. Zusätzlich gibt es bei dem Aufbau immer wieder Probleme, welche die tatsächlich für Unterricht verbleibende Zeit erheblich vermindern. Diese Probleme soll yasp lösen, indem der "USB-Master" in einem Webbrowser inklusive einiger Hardwareelemente wie LEDs, einem Potentiometer und zwei Buttons simuliert wird. Das bedeutet, dass der Aufbau der Hardware entfällt und sich die Vorbereitungszeit auf die Startzeit der Computer beschränkt. Zusätzlich ermöglicht yasp den Schülerinnen und Schülern auch zu Hause weiter zu Programmieren oder effektiver für einen Test zu lernen. Da der zum USB-Master zugehörige Editor sehr simpel gehalten ist, wurde dieser in yasp ebenfalls erweitert, was eine weitere Lernhürde eliminiert. Um den Übergang vom USB-Master zu yasp möglichst einfach zu halten, wurde die selbe Assemblervariante implementiert. Das bedeutet, dass problemlos zu yasp gewechselt werden kann, ohne die Beispielprogramme und Anleitungen anzupassen. Das Projekt hat den USB-Master in der HTBLuVA Spengergasse bereits erfolgreich ersetzt und wird derzeit in allen Assembler-Stunden eingesetzt. Die Rückmeldung von ProfessorInnen und SchülerInnen ist durchgehend positiv, was vor allem auf den Entfall der echten Hardware und den neuen Debugger zurückzuführen ist. Da das Projekt nicht nur für Schülerinnen und Schülern der Spengergasse interessant ist, wurde es auf diversen Blogs und News-Seiten wie "Hack a day", "reddit" und "hackernews" veröffentlicht, was zu einer Gesamtzahl von über 40.000 Besuchern im Frühjahr 2014 auf der Website http://yasp.me/ geführt hat.

Approbation (Datum / Unterschrift)	Prüfer/in	Abteilungsvorstand

htl bildung mit zukunft	**HTBLuVA Wien V**	SPENGERGASSE ausbildung mit zukunft
	Abteilung:	Informatik
	Ausbildungsschwerpunkt:	Intelligente Systeme, Tools und Methodik

DIPLOMA THESIS
Documentation

Author(s)	Michael Lutonsky 5EHIF Robert Fischer 5AHIF
Form Academic Year	5AHIF, 2013/14 5EHIF, 2013/14
Topic	Reimplement the USB-Master in the Web (with debugger, assembler, editor and emulator)
Co-operation partners	Höhere Technische Bundes-Lehr und Versuchsanstalt Spengergasse 20 A-1050

Assignment of tasks	The main goal of yasp is to make learning assembler for students as easy as possible. The HTBLuVA Spengergasse in Vienna is using a piece of custom-made hardware, which was also developed there, to accomplish the same goal. Yasp seeks to replace this hardware by a simulation inside a web browser. This means, that students do not need to set up the hardware at the beginning of every lesson. They can use a reliable platform and therefore have more fun learning Assembler. Furthermore it is possible for students to learn the usage of software and hardware at home, without the need to take the hardware home.

Realization	HTML5 (JavaScript, HTML, CSS), PHP

Results	*yasp* is available at http://demo.yasp.me/.

Abstract (English)	The main goal of yasp is to make learning assembler for students as easy as possible. The HTBLuVA Spengergasse in Vienna is using a piece of custom-made hardware, which was also developed there, to accomplish the same goal. It features a simplified variant of an assembly-language to make it easier to understand. Since it is soldered by hand by one of the professors and his students, it is prone to defects. Also, setting it up properly is not an easy task, which shortens the remaining time for the actual lesson considerably. yasp seeks to fix this problem by simulating the "USB-Master" inside a web browser, including a few simple hardware-elements like LEDs, potentiometer and buttons. This means that the time needed to begin a lesson is shortened to the boot time of the computers. It also enables students to learn and exercise programming in the assembly-language at home. In addition to simple simulating the existing hardware, an improved debugger and editor has been implemented, which, also contributes to a smoother learning experience. To make the migration to yasp easier, the same variant of assembly has been implemented, which makes it possible for teachers to just switch to yasp without the need to prepare new sample-programs and tutorials. The project has successfully replaced the "USB-Master" and is currently being used in all assembly-lessons. The feedback from students and teachers has been great, since they do not have to worry about hardware problems anymore. It is also easier for them to work with the software now, because of the improved debugger and editor. Since this project is not only interesting to students of the HTBLuVA Spengergasse, the project has been publicized at numerous blogs and news-sites like "Hack a day", "reddit" and "hackernews". It has been met with a great response and a total number of hits over 40,000 in early 2014 at the project's website: http://yasp.me/.

Approval (Date / Sign)	Prüfer/in	Abteilungsvorstand

Inhaltsverzeichnis

EINLEITUNG **11**

IST-Zustand 12

SOLL-Zustand 13

GRUNDLAGEN UND METHODEN **14**

Projektantrag 15

Vorstudie 18

Vorgehensweise 21

ERGEBNISSE **23**

Pflichtenheft 23

Inhaltsverzeichnis 25

Änderungsverzeichnis 26

Projektbeschreibung 28

Zielbestimmung 29

Produkteinsatz 30

Entwicklungsumgebung 31

Produktfunktionen und Produktumsetzung 32

Anhänge 41

Anhang 1: Glossar 42

Anhang 2: Befehlsdokumentation 44

Anhang 3: Abnahmeprotokoll 49

Anhang 4: Protokolle und Datenstrukturen 63

Anhang 5: Hardware 75

Anhang 5: Emulatoraktivitätsdiagramme 77

Benutzerhandbuch 79

Abnahme 94

Veröffentlichung 97

Abschlussbericht 101

Anregungen für Folgeprojekte 107

Persönliche Stellungname 109

Styleguides 111

PROJEKTDOKUMENTATION 116

Zeitaufwandsauswertung 117

Milestoneplanung 119

Sprint Protokolle 120

TECHNISCHE DOKUMENTATION 134

Inhaltsverzeichnis 135

yasp-Assembler Tutorial 137

Localisation 142

Instructions 143

Hardware 147

FileDialog 148

Communicator 150

Breadboards 152

Assembler 155

Testing 159

Hardware 161

Emulator 162

Emulator Data 165

Emulator Messages 168

Testprogramme 171

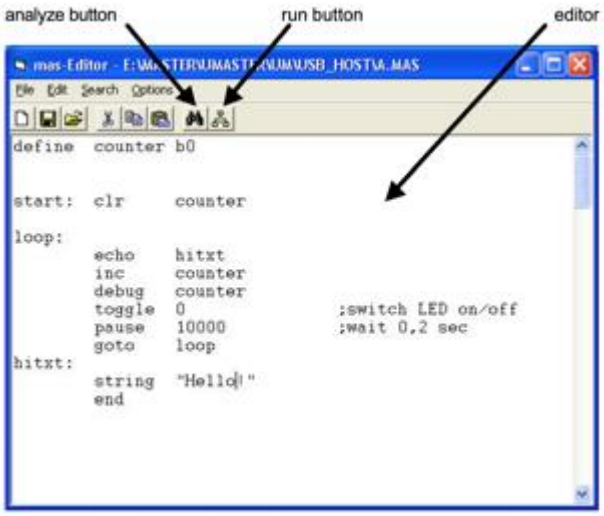
BEFEHLSDOKUMENTATION 182

AUTORENZUORDNUNG 196

Einleitung

IST-Zustand

Schüler und Schülerinnen der ersten und zweiten Klassen der HTBLVA Spengergasse lernen im Unterricht den Umgang mit Assembler und der entsprechenden Hardware. Momentan wird hierfür der USB-Master verwendet. Dies ist eine eigens von Dipl.-Ing. Tarkany entwickelte Hardware auf welcher mit einem vereinfachten Assemblerdialekt programmiert werden kann. Die Programmierung selbst geschieht in dem ebenfalls von Dipl.-Ing. Tarkany geschrieben Programm „mas-editor".



mas-editor // Code-Editor

Der Code-Editor besteht aus einem simplen Textfeld ohne Formatierungshilfen oder Syntaxhighlighting. Das User Interface besteht im Wesentlichen aus zwei Buttons:

⇒ Run

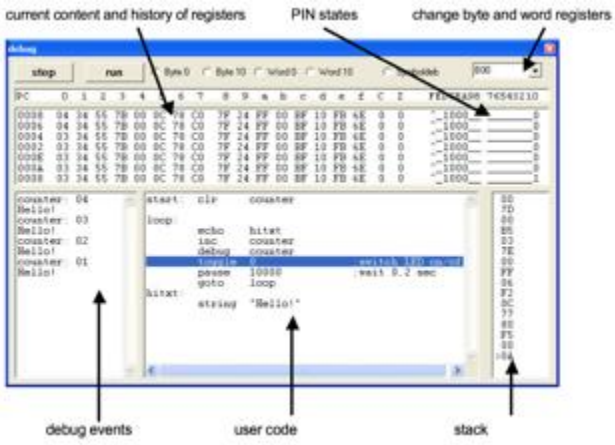
Assembliert den Sourcecode und führt ihn normal auf dem USB- Master aus.

⇒ Analyze

Assembliert den Sourcecode, öffnet den Debugger.

mas-editor // Debugger

Der Debugger zeigt zusätzlich zum Code, welcher momentan auf dem USB-Master läuft, aktuelle und vergangene Registerwerte und den Stack an. Der Step-Button führt den nächsten Befehl aus und hält den USB-Master anschließend wieder an. Der Run-Button startet eine normale Ausführung, welche mit dem Step-Button wieder gestoppt werden kann.



Das Problem an diesen Softwarepaket ist, dass die Hardware auf denen sie basieren (der USB-Master) fehleranfällig und mühsam aufzubauen sind. Zusätzlich ist es den Schülerinnen und Schülern nicht möglich von zuhause aus die Assemblerprogrammierung zu erlernen, weil es nicht gestattet ist die Hardware mit nach Hause zu nehmen.

SOLL-Zustand

Das Ziel von *yasp* ist, den USB-Master samt Entwicklung in den Browser zu verlegen. Dabei soll der Umgang mit Assembler näher gebracht werden. Das bedeutet, dass die Schülerinnen und Schüler sich nicht mehr um das komplizierte Auf- und Abbauen der Hardware kümmern müssen, sie eine zuverlässige Plattform verwenden können und dadurch mehr Spaß beim Entwickeln haben. Weiters können sie auch zu Hause den Umgang mit Hard- und Softwareelementen lernen. Zusätzlich erleichtert es den Unterricht ungemein, da sich niemand Sorgen mehr um das Funktionieren der Hardware machen muss.

Dabei wird das gesamte Software- und Hardwarepaket rund um den USB-Master im Browser implementiert. Es wird eine IDE geben, die mit Syntaxhighlighting und weiteren komfortablen Funktionalitäten das Leben der Schülerinnen und Schüler erleichtern wird. Da der aktuelle Editor keine solchen Funktionalitäten bietet, wird dies das Programmieren mit dem USB-Master um einiges vereinfachen.

Was bringt einem eine ordentliche Entwicklungsumgebung, wenn man seinen Code nicht komfortabel ausführen kann? Genau deshalb wird unser Projekt auch einen entsprechenden Assembler samt Emulator beinhalten. Der Assembler wird dafür zuständig sein, die eingegebenen Befehle zu Binärcode für den Emulator umzuwandeln. Der Emulator nimmt die Bits und Bytes des Assemblers und führt diese dann im Browser aus. Dadurch bleibt alles plattformunabhängig und die Schüler und Schülerinnen können komfortabel den Umgang mit Assembler erlernen.

Der leistungsfähige Debugger vom USB-Master wird natürlich auch Einzug in yasp finden. Die Fähigkeit das aktuell laufende Programm zu stoppen und jeden Winkel des Speichers zu begutachten hilft den Schülerinnen und Schülern die Materie besser zu verstehen und dabei Spaß zu haben.

Grundlagen und Methoden

Projektantrag

DIPLOMARBEIT

5AHIF / 5EHIF – Reife- und Diplomprüfung 2013/14

Thema	Reimplementierung des USB-Masters im Web (samt Debugger, Assembler, Editor und Emulator)
Aufgabenstellung (Kurzfassung)	Das Ziel unseres Projekts ist die Bereitstellung einer plattformunabhängigen Entwicklungsumgebung, die eine komfortable Möglichkeit bietet, den Umgang mit Assembler zu erlernen. Das bedeutet, dass die Schülerinnen und Schüler sich nicht mehr um das komplizierte Auf- und Abbauen der Hardware kümmern müssen, sie eine zuverlässige Plattform verwenden können und dadurch mehr Spaß beim Entwickeln haben. Weiters können sie auch zu Hause den Umgang mit Hard- und Softwareelementen lernen. Zusätzlich erleichtert das den Unterricht der Lehrerinnen und Lehrer sehr, da sie sich keine Sorgen mehr um das Funktionieren der Hardware machen müssen. Ergänzende Ausführungen siehe Projektbeschreibung.

Kandidaten	Schwerpunkt-fach[1]	Betreuer
LUTONSKY Michael	POS	Dipl.-Ing. Grüneis, Dipl.-Ing. Jelinek
FISCHER Robert	MUT	Dipl.-Ing. Grüneis, Dipl.-Ing. Grüneis

Externe Kooperationspartner
Firma / Institution: HTBLVA Spengergasse
Betreuer / Kontaktperson: Dipl.-Ing. Jelinek Robert
Schriftliche Kooperationsvereinbarung liegt vor: Ja
Budget: 100,00€
Bedeckung durch: HTBLVA Spengergasse

Geplante Verwertung der Ergebnisse:
„yasp“ soll den USB-Masters samt Soft- und Hardware ersetzen.

1) Gem. § 23 Abs. 3 Z 3 der Verordnung über die abschließenden Prüfungen in den berufsbildenden mittleren und höheren Schulen, BGBl II Nr. 70/2000.

Beilage 1: Ergänzende Ausführungen

Thema	Reimplementierung des USB-Masters im Web (samt Debugger, Assembler, Editor und Emulator)
Projektmotivation und Aufgabenstellung (Langfassung)	

Projektbeschreibung

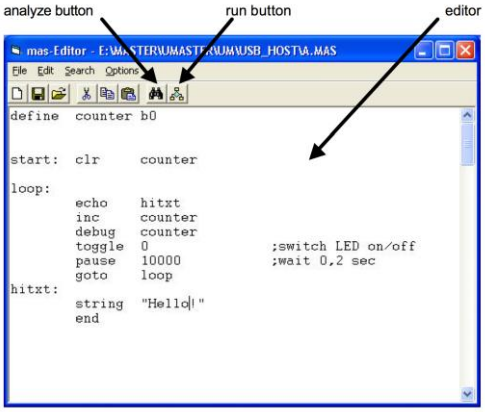
„yasp“ ist ein Assembler/Emulator/IDE/Debugger, der den USB-Master, welcher derzeit in den ersten Klassen als Programmiermittel eingesetzt wird, ersetzen soll.

IST-Situation

Schüler und Schülerinnen der ersten und zweiten Klassen der HTBLVA Spengergasse lernen im Unterricht den Umgang mit Assembler und der entsprechenden Hardware. Momentan wird hierfür der USB-Master verwendet. Dies ist eine eigens von Dipl.-Ing. Tarkany entwickelte Hardware auf welcher mit einem vereinfachten Assemblerdialekt programmiert werden kann. Die Programmierung selbst geschieht in dem ebenfalls von Dipl.-Ing. Tarkany geschrieben Programm „mas-editor“.

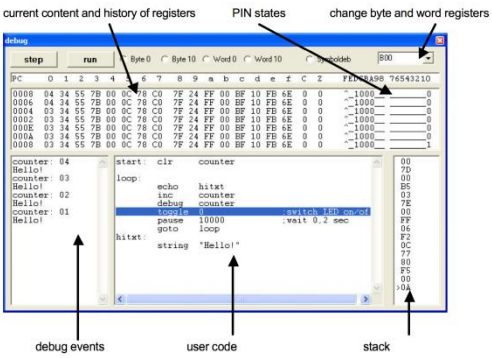
mas-editor // Code-Editor

Der Code-Editor besteht aus einem simplen Textfeld ohne Formatierungshilfen oder Syntaxhighlighting. Das User Interface besteht im Wesentlichen aus zwei Buttons:



⇒ Run
Assembliert den Sourcecode und führt ihn normal auf dem USB- Master aus.

⇒ Analyze
Assembliert den Sourcecode, öffnet den Debugger.



mas-editor // Debugger

Der Debugger zeigt zusätzlich zum Code, welcher momentan auf dem USB-Master läuft, aktuelle und vergangene Registerwerte und den Stack an. Der **Step-Button** führt den nächsten Befehl aus und hält den USB-Master anschließend wieder an. Der **Run-Button** startet eine normale Ausführung, welche mit dem Step-Button wieder gestoppt werden kann.

	SOLL-Situation Das Ziel unseres Projektes ist den USB-Master samt Entwicklung in den Browser zu verlegen. Dabei soll der Umgang mit Assembler näher gebracht werden. Das bedeutet, dass die Schülerinnen und Schüler sich nicht mehr um das komplizierte Auf- und Abbauen der Hardware kümmern müssen, sie eine zuverlässige Plattform verwenden können und dadurch mehr Spaß beim Entwickeln haben. Weiters können sie auch zu Hause den Umgang mit Hard- und Softwareelementen lernen. Zusätzlich erleichtert das das Leben der Lehrerinnen und Lehrer ungemein, da sie sich keine Sorgen mehr um das Wohlergehen der Hardware machen müssen. Dabei wird das gesamte Software- und Hardwarepaket rund um den USB-Master im Browser implementiert. Es wird eine IDE geben, die mit Syntaxhighlighting und weiteren komfortablen Funktionalitäten das Leben der Schülerinnen und Schüler erleichtern wird. Da der aktuelle Editor keine solchen Funktionalitäten bietet, wird dies das Programmieren mit dem USB-Master um einiges vereinfachen. Was bringt einem eine ordentliche Entwicklungsumgebung, wenn man seinen Code nicht komfortabel ausführen kann? Genau deshalb wird unser Projekt auch einen entsprechenden Assembler samt Emulator beinhalten. Der Assembler wird dafür zuständig sein, dass aus den eingegebenen Befehlen Binärcode für den Emulator erstellt wird. Der Emulator nimmt die Bits und Bytes des Assemblers und führt diese dann im Browser aus. Dadurch bleibt alles plattformunabhängig und die Schüler und Schülerinnen können komfortabel den Umgang mit Assembler erlernen. Der leistungsfähige Debugger vom USB-Master wird natürlich auch Einzug in yasp finden. Die Fähigkeit das aktuell laufende Programm zu stoppen und jeden Winkel des Speichers zu begutachten hilft den Schülerinnen und Schülern die Materie besser zu verstehen und dabei Spaß zu haben.
Projektablauf und Meilensteine	• Assembler und Emulator fertig • Editor und Debugger fertig • Benutzer-Dokumentation
Ressourcen	Vom Projektteam bereit gestellt: • GitHub Education Private Repository • OnTime Vom Projektpartner bereit gestellt: • USB Master Dokumentation • Lastenheft • USB-Master Hardware/Software • Testprogramme • WebStorm • Subdomain / Server
Risiken	• JavaScript Engines zu langsam • Ausfall eines Teammitglieds • Unvollständige USB-Master Dokumentation • Neue Browserversionen werden inkompatibel aufgrund veränderter Standards

Vorstudie

Lastenheft

Das Projekt wurde von Dipl.-Ing. Robert Jelinek mit seinem Lastenheft initiiert. Seine Wünsche wurden entsprechend besprochen und die tatsächlich zu implementierenden Punkte finden sich im Pflichtenheft wieder.

JS-Master

Ziel: Eine Simulation für ein Mikroprozessorsystem (System on a Chip - SoC) soll programmiert werden.

Programmierumgebung: objektorientiertes Javascript ohne Toolboxen mit HTML5 und CSS

Beschreibung:

Es ist eine HTML-Seite mit Javascript zu entwerfen, welche ein einfaches Mikroprozessorsystem simuliert.

Randbedingungen:

- Darstellung:
  - Eingabefenster für den Quellcode
  - Ausführungsfenster
  - Simulation der Hardware (LEDs, Tasten, Potentionmeter)
- Quellcode:
  - Eingabe des Quellcodes über einfache Textbox oder Attribut „contenteditable“
  - Einfärben des Quellcodes (Labels, Mnemonics, Parameter, Kommentare)
  - Umwandlung des Quellcodes in Assembler
  - Laden und Speicher von Quellcode über PHP
- Prozessorsimulation (beim Start auswählbar aus mehreren Architekturen):
  - Echte Simulation mit Speicher und Codes
  - Eigener Prozessor mit 8 I/O-Pins und zwei analogen Eingängen
  - Intel-Architektur (16 / 32 Bit)
  - ARM-Architektur
- Register des eigenen Prozessors:
  - 32 mal Byte (8Bit)
  - 16 mal Word (16 Bit)
- Befehle des eigenen Prozessors:
  - Einfache Adressierungsarten
  - Kopieren Register, Hauptspeicher
  - Arithmetisch – Logisch – Bitschieben, Vergleich
  - bedingte und unbedingte Sprünge, absolut und relativ
  - Unterprogrammaufrufe
  - I/O
  - Interruptverarbeitung, Tabelle ab Adresse 0x100, je 2 Byte pro Adresse
- Ausführungsfenster:
  - Darstellung von Code, Registern, Stack, Status, PC, I/O und Interrupts
  - Veränderbarkeit von Registern, Stack, Status, PC, I/O und Interrupts
  - Setzen und Löschen von Haltepunkten mit Durchgangszählern
  - schrittweise Ausführung
- Simulation der Hardware:
  - LED wird je nach Anteil des Leuchtens in ihrer Helligkeit simuliert (PWM)
  - Simulation eines Schrittmotors
- Javascript:
  - Ausführung in Firefox, Safari, Android Browser, Opera, Internet Explorer
  - Objekte für simulierte Hardwareelemente und Darstellungselemente
  - keine Toolboxen für Javascript

SCRUM-Tools

Als Projektentwicklungsmodell wurde während der Entwicklung von *yasp* auf SCRUM gesetzt. Konventionelle Projektverwaltungswerkzeuge, wie das von der Schule zur Verfügung gestellte MS-Project, unterstützen diese Form des Projektmanagements unzureichend, weshalb wir gezwungen waren eine Alternative zu suchen.

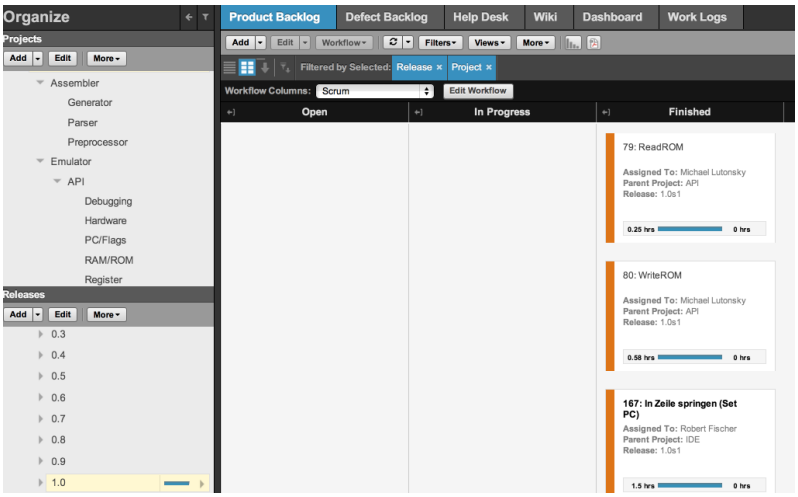
Untersuchung

Im Rahmen der Vorstudie haben wir mehrere SCRUM-Tools auf ihre Funktionen untersucht. Eine der harten Voraussetzungen für das Tool waren, keine beziehungsweise nur geringe Kosten, da die Schule das notwendige Budget nicht zur Verfügung stellen konnte. Zusätzlich muss das Tool eine Exportfunktion besitzen, da wir diese für die Zeitauswertung benötigen. Die Ergebnisse unserer Recherche finden sich in folgender Tabelle.

Name	Kosten	Plattform	Fehlende Funktionen
Targetprocess	Gratis	Web	Mangelhaftes Sprintmanagement keine Burndowncharts
agilefant	Gratis	Web	Eigener Server notwendig
pivotaltracker	€ 7,00 / Monat	Web	Fehlendes Timetracking kostenpflichtig
sprintly	€ 17.34 / Monat	Web	Mangelhaftes Sprintmanagement kostenpflichtig
Scrumdesk	Gratis	Desktop	Fehlender Export Desktopanwendung
ScrumDo	€ 14.44 / Monat	Web	Kostenpflichtig
BananaScrum	Gratis	Web	Mangelhaftes Sprintmanagement Fehlendes Export
TeamBox	Gratis	Web	Keine Burndowncharts Mangelhaftes Sprintmanagement

OnTimeNow

von *axosoft*: http://www.axosoft.com/



Letztendlich haben wir uns für OnTime von axosoft entschieden. Dieses Tool bietet nicht nur eine ausgezeichnete Online Oberfläche und Plattformunabhängigkeit, eine Exportfunktion, sondern erfüllt auch alle weiteren Kriterien:

- Zeitaufzeichnung
- Sprint Management
- Mehrbenutzersystem
- Issuetracker

Dieses Projektmanagementwerkzeug war perfekt für *yasp* geeignet. Ein kostenfreier Account wurde uns von *axosoft* zur Verfügung gestellt.

Risiken

Risiko	Wahrscheinlichkeit	Auswirkung	Maßnahme
JavaScript-Engines zu langsam	Hoch	Mittel	Optimierung
Ein Teammitglied fällt aus	Gering	Hoch	Projekt kürzen
Unvollständige USB-Master Dokumentation	Hoch	Gering	Informieren

Technologien und Frameworks

Realisierung

- Client: HTML5, JavaScript und CSS
- Server: PHP
- Testing: QUnit
- User Interface: Bootstrap, jQuery
- Editor: CodeMirror
- Icons: Font-Awesome
- Befehlsdokumentation: JSDoc + doctool (Eigene Entwicklung)
- Task-Runner: grunt
- Packet Manager: npm

Projektentwicklung

- Versionsverwaltung: GitHub
- Projektmanagement: OnTimeNow
- Entwicklungsumgebung: WebStorm

Vorgehensweise

Projektmanagement

Zur Organisation des Projektes haben wir eine für den Schulgebrauch angepasste Version von SCRUM verwendet. Die User-Stories wurden alle vor Projektbeginn erstellt, gerankt und in Kategorien (zum Beispiel "Emulator" oder "Assembler") eingeteilt. Da das Projekt mit zwei Mitgliedern sehr klein war und es zu diesem Zeitpunkt ein ausführliches Pflichtenheft gab, haben wir auf eine genauere Beschreibung der Stories verzichtet und es nur bei einem Titel belassen. Zum Beispiel ist die Story "Registers View" (Kategorie: IDE / Debugger) nur im Pflichtenheft anhand eines Mockups näher beschrieben, was im nachhinein betrachtet völlig ausgereicht hat.

SCRUM-Tool: OnTimeNow

Wie in der Vorstudie beschrieben, haben wir mehrere SCRUM-Tools auf die Tauglichkeit für unser Vorhaben untersucht. Letztendlich haben wir uns für "OnTimeNow" von axosoft entschieden, welches ein Web-Tool ist. Ein Account für zwei Benutzer wurde uns von axosoft kostenlos für Schulzwecke zur Verfügung gestellt. Im Großen und Ganzen waren wir sehr zufrieden mit dem Tool, allerdings lassen die Export-Möglichkeiten zur weiteren Auswertung ein wenig zu wünschen übrig. Ein weiterer kleiner Kritikpunkt ist, dass die Änderungen anderer Nutzer nicht sofort bei allen offenen Instanzen der Software sichtbar sind. Abhilfe schafft der "Reload"-Button auf der Seite der auch wirklich nur die betreffende Liste neu lädt, nicht die ganze Seite.

Sprints

Unsere Sprint-Länge haben wir auf eine Woche fixiert, da eine so kurze Länge recht wenig Controlling innerhalb des Sprints selbst erfordert. Am Ende jedes Sprints wurden alle erledigten Features getestet und anschließend in die stabile Version miteinbezogen (siehe "Verwaltung des Source-Codes"). Weiters haben wir nach jedem Sprint ein Sprintprotokoll verfasst, das den entsprechenden Sprint dokumentiert hat (Siehe "Sprintprotokolle"). Unser SCRUM-Tool hat zusätzlich ein Task Board und Time-Tracking eingebaut, so dass wir die komplette Abwicklung der Sprints in OnTimeNow erledigen konnten ohne ein zweites Tool zu verwenden.

Verwaltung des Source-Codes

Sourcecode-Hosting

Für das Hosting des Sourcecodes haben wir uns für GitHub entschieden, da das Unternehmen kostenfreie, private Accounts für Bildungszwecke zur Verfügung stellt. Auf Grund der kleinen Teamgröße und damit verbundenen kleineren Code-Menge pro Sprint haben wir uns für ein simples Branchingmodel, mit nur zwei Branches entschieden: "master" und "stabile". Der master-Branch stellt den aktuellen Entwicklungsstand dar und ist möglicherweise nicht funktionstüchtig. Hier findet auch die Entwicklung der Sprints statt. In Kontrast dazu steht der stabile-Branch, welcher immer den Stand nach dem letzten Sprint enthält. Dieser bietet immer eine funktionierende Version von yasp an und sollte vor allem von Endnutzern verwendet werden.

Kommunikation

Vor Projektbeginn fanden fast ausschließlich Meetings statt, auf denen das Pflichtenheft und die User-Stories geschrieben wurde. Nach jedem Meeting gab es ein bis zwei kleine Aufgaben für jedes Mitglied, die bis zum nächsten zu erledigen waren.

Die Kommunikation während des Projekts wurde großteils über das Internet abgewickelt, da die beiden Teammitglieder in verschiedenen Schulklassen waren. Hier wurde meistens die Software "Skype" verwendet, um auch Telefonkonferenzen zu ermöglichen. Alle Dokumente die während des Projektes anfielen wurden in Google Docs erfasst, da dies eine Echtzeitkollaboration möglich macht.
Am Ende jedes Sprints haben wir uns getroffen, die Erneuerungen besprochen und anschließend den stabile-Branch aktualisiert. Hier wurden auch "Defects" (Fehler) in OnTime eingetragen um Nachbesserungen zu organisieren. Die Protokolle jedes Meetings wurden an beide Betreuer ausgeschickt, und sind im Anhang.

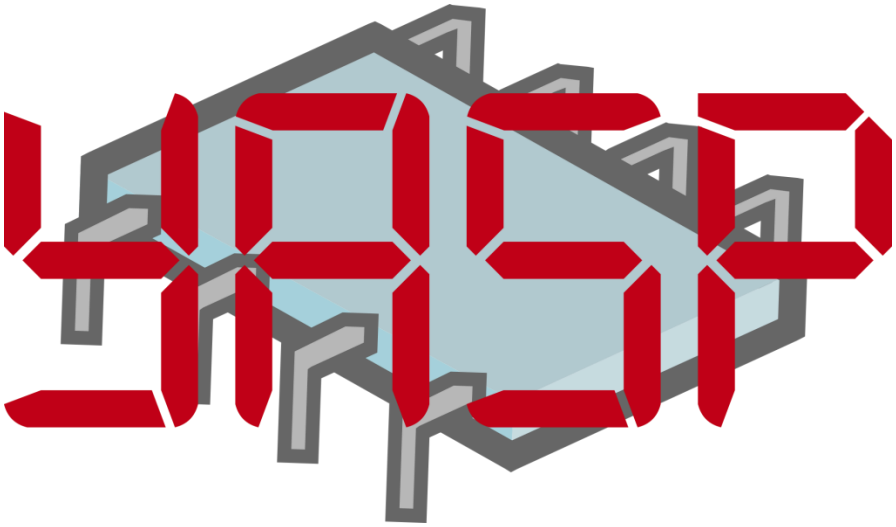
Automatische Tests

Als Unit-Test Framework wurde JUnit verwendet, auf welches wir allerdings viele Erweiterungen aufsetzten um das Testen der Instruktionen oder Breakpoints zu vereinfachen. Mehr Informationen hierzu finden sich im Anhang "Technische Dokumentation", welche auch im Internet auf GitHub verfügbar ist. Wie auch im Abschlussbericht erwähnt haben wir keinen Continuous Integration-Server eingesetzt, was im Nachhinein gesehen ein Nachteil war, da manchmal Fehler nicht sofort aufgefallen sind.

JSDoc

Zur Dokumentation der einzelnen JavaScript-Funktionen und Felder in yasp haben wir JSDoc eingesetzt. Da JavaScript die Angabe von Typen im Sourcecode nicht vorsieht, wurden die erwarteten Parametertypen jeder Funktion ebenfalls in JSDoc notiert. Das Tool ermöglicht es, einen HTML-Export der Dokumentation zu erstellen, welcher im GitHub Repository zu finden ist.

Ergebnisse



Pflichtenheft

des Diplomprojektes an der HTLBVA Spengergasse

Autoren:
Robert Fischer (*kurz: fish*)
Michael Lutonsky (*kurz: luto*)

Inhaltsverzeichnis

Änderungsverzeichnis **26**

Projektbeschreibung **28**

Zielbestimmung **29**
Soll-Ziele 29
Kann-Ziele 29
Nicht-Ziele 29

Produkteinsatz **30**
Anwendungsbereiche 30
Zielgruppen 30
Betriebsbedingungen 30

Entwicklungsumgebung **31**

Produktfunktionen und Produktumsetzung **32**
Komponentenkommunikationsübersichtsdiagramm 32
Produktdaten 33
Produktleistungen 33

Anhänge **41**

Anhang 1: Glossar **42**

Anhang 2: Befehlsdokumentation **44**
Ein Befehl 44
Befehlsformat 44
`Exec`-Funktion 45
Emulator-Architektur 45
Instruktionen 46
Direktiven 47

Anhang 3: Abnahmeprotokoll **49**
Debugger 59

Anhang 4: Protokolle und Datenstrukturen **63**
Communicator 63
Assembler 65
Emulator 66
Server 74

Anhang 5: Hardware **75**
Beschreibung 75
Methoden 75
Hardwaretypen 75
Hardwareeinsatz 76

Anhang 5: Emulatoraktivitätsdiagramme **77**
Generelle Funktionsweise 77
Action-Behandlung 78

Änderungsverzeichnis

Name	Datum	Autor	Änderungen
Draft 1	10. August 2013		Erste Version
Draft 2	11. August 2013	luto	Änderungen in Anhang 4 • von "Workerprotokoll" in "Protokolle und Datenstrukturen" umbenannt • Diverse Layout-Fixes • Aufteilung in Datenstrukturen und Protokoll • Map in `ASSEMBLE`-Action des Assemblers hinzu
Draft 3	12. August 2013	fish	Änderungen in Anhang 4 • Ausbesserung von Rechtsschreib- und Inhaltsfehlern • Datenstruktur Breakpoint beschrieben • Datenstruktur State beschrieben • Broadcasts (IO_Changed / Started/Stopped) • Neuer Befehl `SET_IO`
Draft 4	13. August 2013	fish	Änderungen in Anhang 4 • Datenstruktur Condition beschrieben • `BREAK`, `SET_BREAKPOINTS` beschrieben • Rechtsschreibfehler ausgebessert
Draft 5	14.August 2013	fish	Änderungen im Pflichtenheft • Produktleistungen / IDE • Angefangen: Nichtfunktionale Anforderungen / Datenschutz
Draft 6	15. August 2013	fish luto	Änderungen im Pflichtenheft • Kleinere Anpassungen bei IDE • Server & Datenmodell • Use Case Diagramm IDE • Komponentenkommunikationsübersichts-Diagramm • Mockup IDE • Emulator Aktivitätsdiagramme • Hardware: Klassendiagramm & Infos • Mockup Debugger Änderungen in Anhang 4, Protokolle/Datenstrukturen • Server

Draft 7	12. September 2013	luto	Titelblatt Betriebsbedingungen angepasst Projektziele angepasst Befehlsdokumentation, Formatänderungen Entwicklungsumgebung
Draft 8	13. September 2013	fish	Debugger Mockups Redesign Debugger Beschreibung
Draft 9	14. September 2013	luto fish	Anpassung der KANN/SOLL-Ziele Abnahmeprotokoll
Draft 10	19. September 2013	luto fish	Assembler Styleguides
Draft 11	22. September 2013	luto fish	Definition Emulator Architektur Anpassung Befehlsdokumentation Definition Hardware Erweiterung des Abnahmeprotokolls Glossar erweitert
Draft 12	23. September 2013	fish	Serverteil ausgebaut und angepasst Assembler Pass Beschreibung Rechtsschreib und Grammatikkorrekturen
Draft 13	25. September 2013	luto	layout fixes
Final	26. September 2013	luto fish	Finale Version.

Projektbeschreibung

Das Ziel von *yasp* ist, den USB-Master samt Entwicklung in den Browser zu verlegen. Dabei soll der Umgang mit Assembler näher gebracht werden. Das bedeutet, dass die Schülerinnen und Schüler sich nicht mehr um das komplizierte Auf- und Abbauen der Hardware kümmern müssen, sie eine zuverlässige Plattform verwenden können und dadurch mehr Spaß beim Entwickeln haben. Weiters können sie auch zu Hause den Umgang mit Hard- und Softwareelementen lernen. Zusätzlich erleichtert es den Unterricht ungemein, da sich niemand Sorgen mehr um das Funktionieren der Hardware machen muss.

Dabei wird das gesamte Software- und Hardwarepaket rund um den USB-Master im Browser implementiert. Es wird eine IDE geben, die mit Syntaxhighlighting und weiteren komfortablen Funktionalitäten das Leben der Schülerinnen und Schüler erleichtern wird. Da der aktuelle Editor keine solchen Funktionalitäten bietet, wird dies das Programmieren mit dem USB-Master um einiges vereinfachen.

Was bringt einem eine ordentliche Entwicklungsumgebung, wenn man seinen Code nicht komfortabel ausführen kann? Genau deshalb wird unser Projekt auch einen entsprechenden Assembler samt Emulator beinhalten. Der Assembler wird dafür zuständig sein, die eingegebenen Befehle zu Binärcode für den Emulator umzuwandeln. Der Emulator nimmt die Bits und Bytes des Assemblers und führt diese dann im Browser aus. Dadurch bleibt alles plattformunabhängig und die Schüler und Schülerinnen können komfortabel den Umgang mit Assembler erlernen.

Der leistungsfähige Debugger vom USB-Master wird natürlich auch Einzug in yasp finden. Die Fähigkeit das aktuell laufende Programm zu stoppen und jeden Winkel des Speichers zu begutachten hilft den Schülerinnen und Schülern die Materie besser zu verstehen und dabei Spaß zu haben.

Zielbestimmung

Soll-Ziele

1. Assembler implementieren
2. Emulator implementieren
    a. CPU (gemäß der in *Anhang 2, Befehlsdokumentation* definierten Befehle)
    b. RAM
    c. Hardware (gemäß *Anhang 5, Hardware*)
3. Debugger implementieren
4. Integrierte Entwicklungsumgebung im Browser
5. Modularer Aufbau, so dass verschiedene Architekturen (ARM, x86/64, MIPS, etc.) in Folgeprojekten implementiert werden können.
6. Clientseitiges Speichern von Quellcode im *localStorage*

Kann-Ziele

1. Mehrbenutzersystem mit gleichzeitigem Editieren desselben Programmes
2. C-Like-Compiler mit Echtzeit Assembler Darstellung
3. Unterstützung mobiler Endgeräte (z.B.: Android oder iOS)
4. Serverseitiges Speichern von Quellcode
5. Unterstützung der Sound-Befehle
6. Unterstützung von Autovervollständigung in der IDE

Nicht-Ziele

1. Implementation einer anderen Zielplattform (z.B.: x86, ARM, usw.)
2. Unterstützung des RaspberryPis als Zielplattform (Realisierung in Folgeprojekt)
3. Schulung der Nutzer

Produkteinsatz

Das Produkt wird als Lehrmittel an der HTBLVA Spengergasse eingesetzt werden.

Anwendungsbereiche

- Lernen und verstehen des Umgangs mit der USBMaster Plattform
- Kosten- und Aufwandseinsparung durch Wegfallen des Hardwareeinsatzes

Zielgruppen

- Schüler der 1. und 2. Klassen
- Lehrer der selbigen

Betriebsbedingungen

yasp wird in den folgenden Browsern getestet:

- Chrome 32
- Firefox 26
- Internet Explorer 10

Die Funktionalität wird in den oben genannten Browsern, auf für das Jahr 2013 gängiger Hardware, getestet. Die Lauffähigkeit in anderen Umgebungen kann nicht sichergestellt werden.

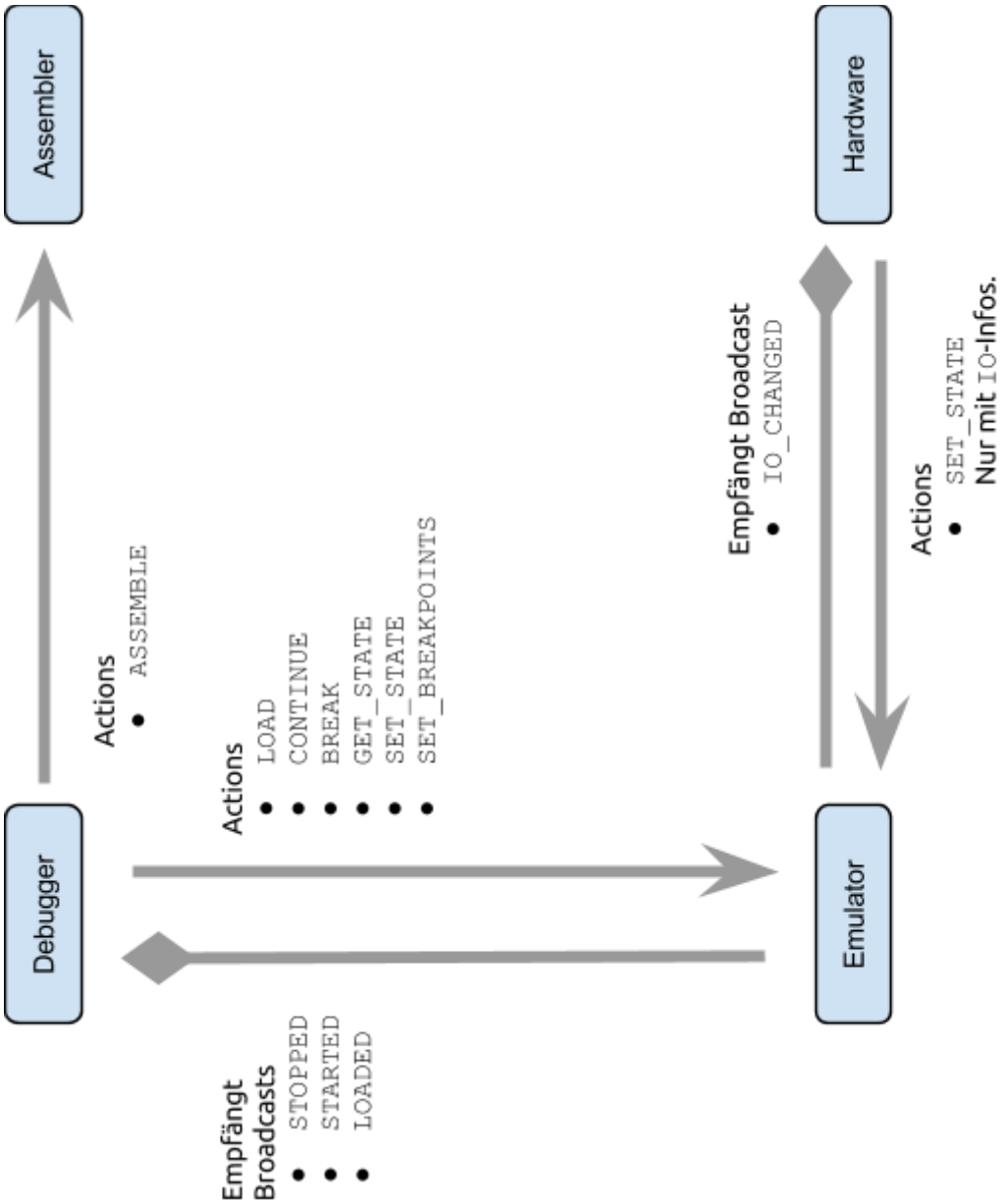
Entwicklungsumgebung

Für die Entwicklung wird unsererseits Linux und OS X verwendet. Wir werden darauf achten, dass alle verwendeten Tools (z.B.: grunt, less, qunit, …) auch mit anderen Betriebssystemen kompatibel sind können aber hier keinen Support bieten.
Bei der Übergabe wird ein *Build* und Quellcode zur Verfügung gestellt, welcher den letzten, fertigen Stand unserer Entwicklung darstellt.

Produktfunktionen und Produktumsetzung

Komponentenkommunikationsübersichtsdiagramm

Dieses Diagramm beschreibt wie die einzelnen Komponenten miteinander kommunizieren. Informationen zur genauen technischen Umsetzung dieser *Actions* und *Broadcasts* findet sich im *Anhang 4, Protokolle und Datenstrukturen*.



Produktdaten

Typ	Ort	Zugriffsart
Assemblercode	localStorage oder Server	lesen / schreiben
Maschinencode	RAM	lesen / schreiben
Befehlsdefinitionen	JavaScript Code	lesen

Produktleistungen

Debugger

Funktionen

- Step In
  Führt die nächste Instruktion aus und pausiert danach.
- Step Back *KANN*
  Versetzt den Emulator in den Zustand den er vor der Ausführung der letzten Instruktion hatte.
- Step Out *KANN*
  Verhält sich wie Step In, allerdings wird ausgeführt bis die nächste `RETURN` Instruktion ausgeführt wird.
- Break
  Stoppt das Programm beim aktuellen Ausführungspunkt
- Continue
  Die Programmausführung geht normal weiter
- Set Program Counter
  Man wählt eine Zeile aus und der Debugger springt zur angegebenen
- Execute *KANN*
  Führt beliebigen Assemblercode an der aktuellen Stelle aus
- Breakpoints
  Der Debugger hält an, sobald die Ausführung diesen Befehl erreicht.
  - Setzen
  - Entfernen
  - Temporär deaktivieren
  - Bedingte Breakpoints *KANN*
    Hält an, wenn die eingegebene Bedingung erfüllt ist.
  - Zeilenbezogene bedingte Breakpoints *KANN*
    Hält an, wenn die Bedingung erfüllt ist und der Emulator die angegebene Zeile ausführt.

Externe Kommunikation

Der Debugger kommuniziert mit Emulator und dem Assembler.
Details siehe *Anhang 4, Protokolle und Datenstrukturen*

Assembler
Die Hauptaufgabe des Assemblers ist es Assemblercode in Maschinencode zu übersetzen. Hierbei gibt es folgende Resultate, neben dem Maschinencode:

- Symboltabelle
- Map

Das Format und die genauen Inhalte dieser sind in *Anhang 2, Befehlsdokumentation* näher definiert. Welche Befehle vom Assembler in welcher Weise umgesetzt werden ist in *Anhang 2, Befehlsdokumentation*, definiert.

Die Assemblierung einer Quelldatei unterteilt sich in folgende Schritte, die der Reihe nach ausgeführt werden:

1. Lexikalische Analyse: Zerteilt den Quellcode in einzelne Tokens.
2. Syntaktische Analyse: Überprüft die syntaktische Korrektheit
3. Semantische Analyse: Erstellt die Map *Siehe Anhang 4, Protokolle und Datenstrukturen*
4. Präprozessor: Ersetzt die Direktiven
5. Parser: Erkennt anhand der Tokens die Befehle und erstellt eine entsprechende Darstellung im Speicher
6. Generator: Generiert anhand der zuvor in den Speicher geladenen Befehlen den Maschinencode der schließlich vom Emulator ausgeführt werden können.

Externe Kommunikation
Siehe *Anhang 4, Protokolle und Datenstrukturen*.

Emulator
Der Emulator führt den vom Assembler generierten Maschinencode aus. Er läuft in einem eigenen *Web Worker*, was eine ruckelfreie Entwicklungsumgebung für die User schafft. Für weitere Informationen bezüglich unterstützter Befehle siehe *Anhang 2, Befehlsdokumentation*. Ein Aktivitätsdiagramm welches die genaue Funktionsweise des Emulators erklärt befindet sich im *Anhang 6, Emulator Aktivitätsdiagramme*.

Interrupts
yasp unterstützt Interrupts.
Siehe Befehle `DA`, `ORG`, `INT`, `ENABLE`, `DISABLE` in *Anhang 2, Befehlsdokumentation.*

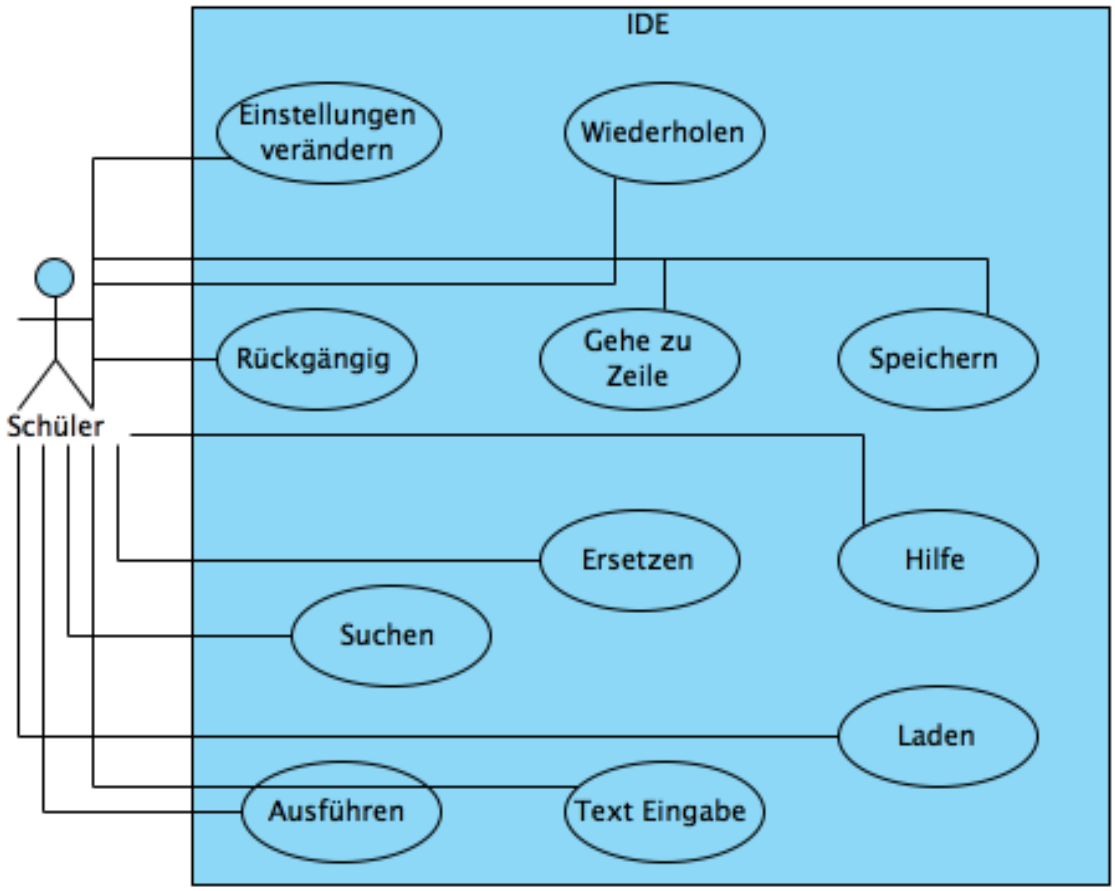
Externe Kommunikation
Siehe *Anhang 4, Protokolle und Datenstrukturen*.

IDE
Die integrierte Entwicklungsumgebung von *yasp* wird in einer HTML5 Oberfläche implementiert sein. Dabei wird das UI Toolkit *Bootstrap* verwendet, da dieses sehr flexibel und leistungsstark ist. Als Texteditor mit Syntaxhighlighting Funktion verwenden wir Code Mirror, da es die Bedürfnisse von *yasp* am besten erfüllt, sehr gut dokumentiert ist und es sich in anderen Projekten bereits gut bewährt hat.

Die IDE wird aus einem Textfeld mit Syntaxhighlightingfunktion und einer Werkzeugleiste bestehen. In das Textfeld wird der Assemblercode eingegeben und die Werkzeugleiste ermöglicht weitere Operationen:

- Ausführen: Führt das Programm aus und startet den Debugger
- Speichern: Speichert den aktuellen Source Code im `localStorage` oder auf einem Server
- Laden: Lädt einen Source Code vom `localStorage` oder Server

- Hilfe: Öffnet die Hilfe mit einer Erklärung zu jedem Befehl und zur generellen Funktionsweise
- Einstellungen: Festlegen folgender Optionen
  - Syntaxhighlighting Themes
  - Wie viele Leerzeichen ein Tab hat
  - Automatisches speichern Ja/Nein
  - Autovervollständigung Ja/Nein
  - Sprache Englisch
- Rückgängig / Wiederholen
- Suchen / Ersetzen / Gehe Zu Zeile
- Codeformatierung (gemäß Styleguide siehe *Anhang 7, Assembler Styleguide*)
  - Automatische Formatierung *KANN*



Externe Kommunikation
Siehe *Anhang 4, Protokolle und Datenstrukturen.*

Server

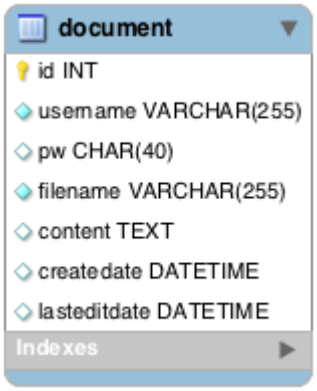
Ein Anwender kann den geschriebenen Quellcode nicht nur im `localStorage` speichern, sondern auch auf einem Server. Über den Speichern Dialog kann man wahlweise die Quelldatei auf dem Server speichern. Hierbei hat man folgende Optionen:

- Benutzername + Passwort: Benutzer können nur auf die Datei zugreifen, sofern das Passwort richtig ist bzw. das Masterpasswort angegeben wurde.
- Benutzername + Kein Passwort: Da vielleicht Schülerinnen und Schüler ihr Ergebnis mit anderen Schülern teilen wollen oder Angst haben ihr Passwort zu vergessen, wird es die Möglichkeit geben, Dateien ungeschützt zu speichern.

Über den *Laden* Dialog wird die entsprechende Datei vom Server geladen. Dabei bekommt man nach Angabe des Benutzername eine Auswahl der unter diesem Namen gespeicherten Dokumente. Diese kann man schließlich nach Angabe des Passworts (sofern benötigt) öffnen. Weiters haben Lehrer nach Angabe des Masterpasswortes die Möglichkeit beliebige Dateien zu öffnen.

Die Dateien der Benutzer werden in einer filebasierten Datenbank gespeichert, die via REST API angesprochen wird. Die Serverprogrammierung wird mithilfe von PHP umgesetzt. Für weitere Informationen, siehe REST API, definiert in *Anhang 4, Protokolle und Datenstrukturen*.

Das Datenmodell sieht folgendermaßen aus, kann allerdings während der Entwicklung modifiziert werden):

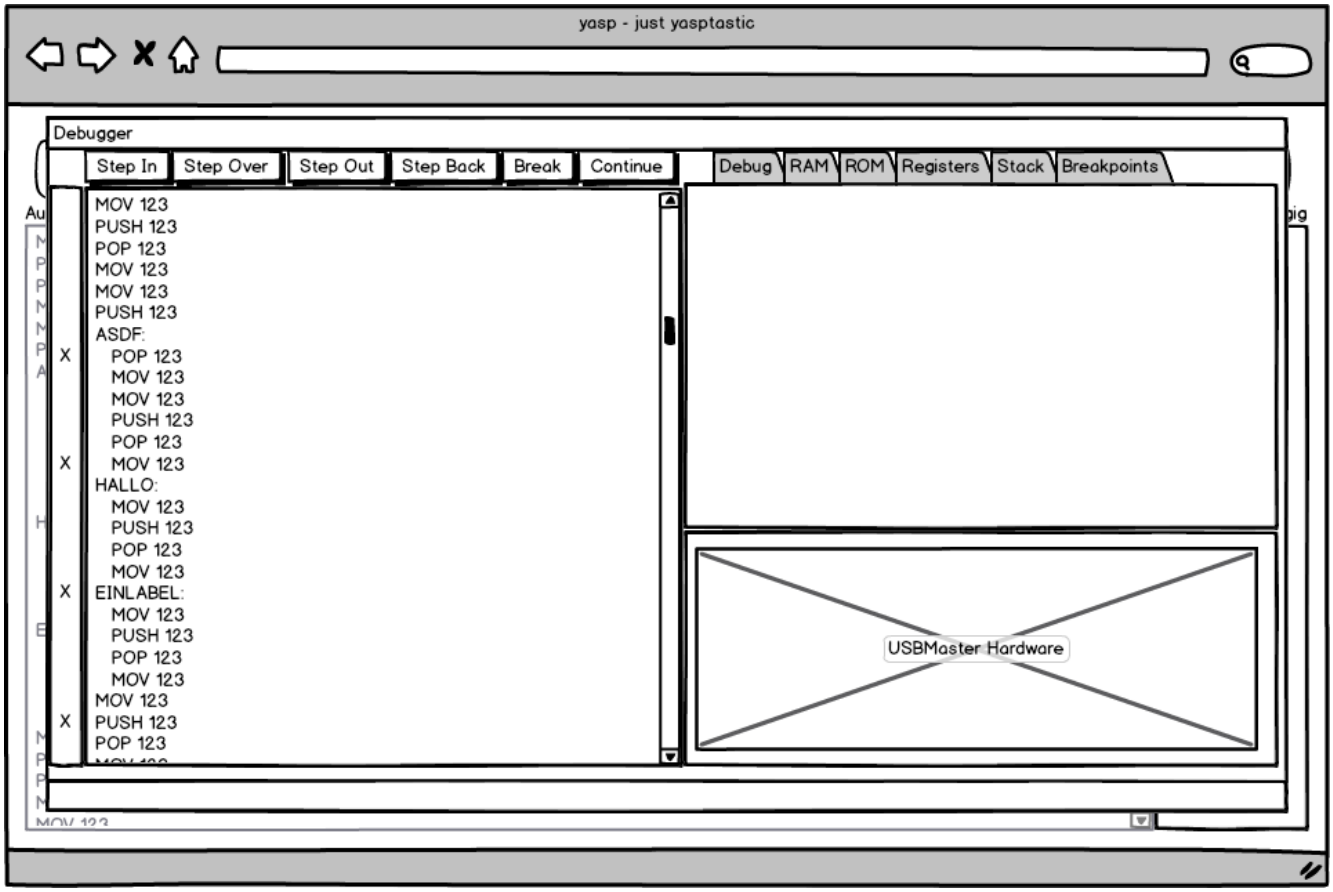


- id: ist eine Numerische Identifikationsnummer die automatisch vergeben wird
- username: Unter jenem Usernamen wurde die Datei gespeichert
- filename: Der Dateiname
- content: Die Quelldatei
- createdate: Datum als die Datei erstellt wurde
- lasteditdate: Datum als die Datei zuletzt geändert wurde.

Benutzeroberfläche

Debugger

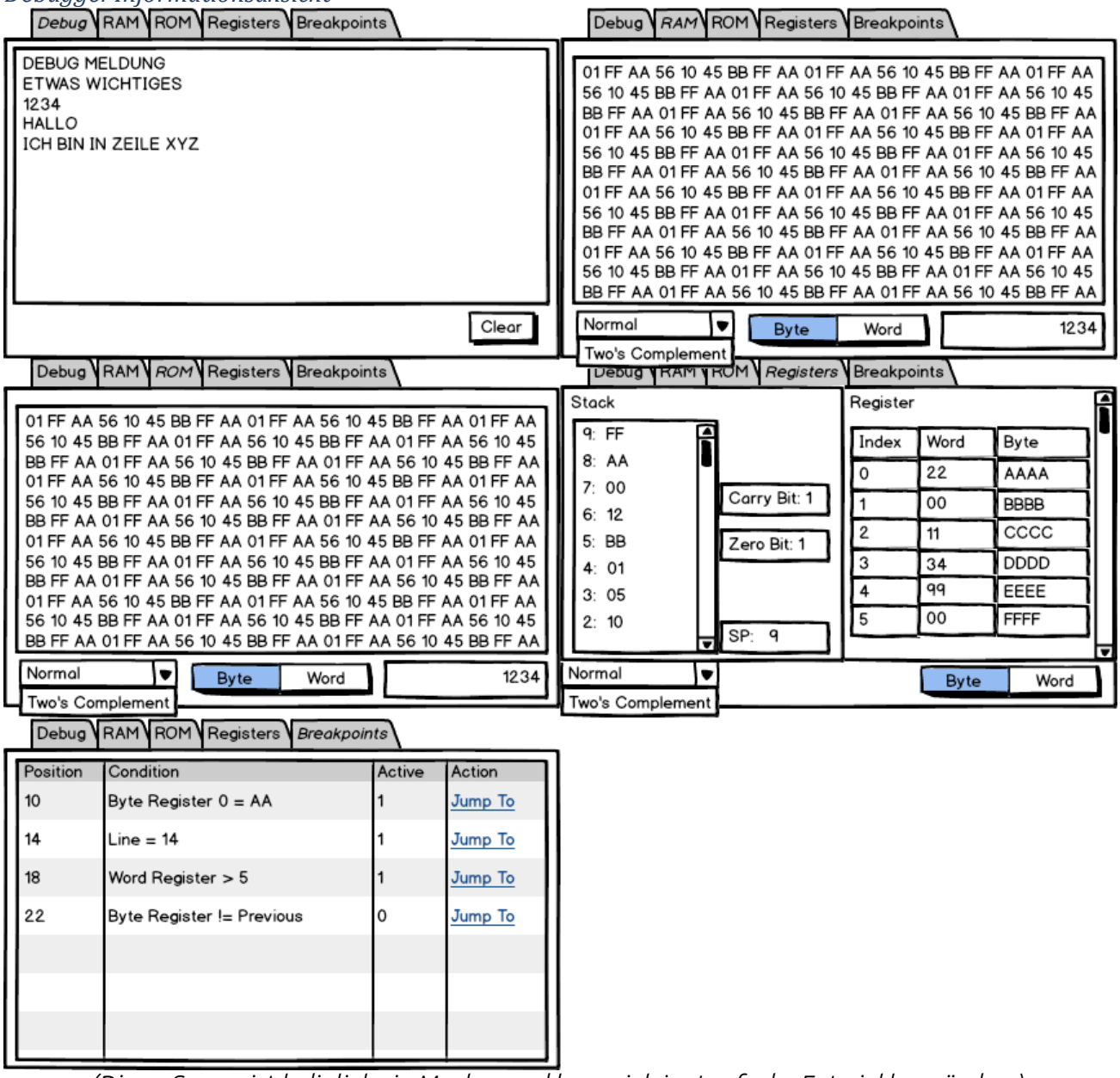
Der Debugger erscheint als Dialog, sobald man auf den *Ausführen*-Button in der IDE klickt.



(Dieser Screen ist lediglich ein Mockup und kann sich im Laufe der Entwicklung ändern)

- Links ist das Codepanel, das nicht editierbar ist.
- Über dem Codepanel sind die Debuggingaktionen (Step In, Step Back, Continue, …)
- Rechts oben davon ist die Informationsansicht, die den aktuellen Status ausgibt. Für weitere Informationen siehe entsprechende Mockups.
- Rechts unten von dem Codepanel ist eine Ansicht die den aktuellen Hardwarestatus ausgibt.

Debugger Informationsansicht



(Dieser Screen ist lediglich ein Mockup und kann sich im Laufe der Entwicklung ändern)

Debug

Gibt jene Werte aus, die als Parameter an Debug-Befehle übergeben wurden. Dabei werden die Befehle sequentiell in einer uneditierbaren Textbox ausgegeben werden. Der clear-Button leert die Textbox wieder. Die Textbox wird zusätzlich automatisch beim Starten geleert.

RAM

Gibt den Inhalt des RAMs aus. In der unteren Leiste kann man auswählen in welchem Format der Inhalt ausgegeben werden soll. Außerdem kann man den Inhalt eines Bytes/Words setzen, indem man mit dem Cursor draufklickt und dann in die untere Textbox den Inhalt eingibt.

ROM

Siehe RAM. Zeigt statt dem RAM den ROM an.

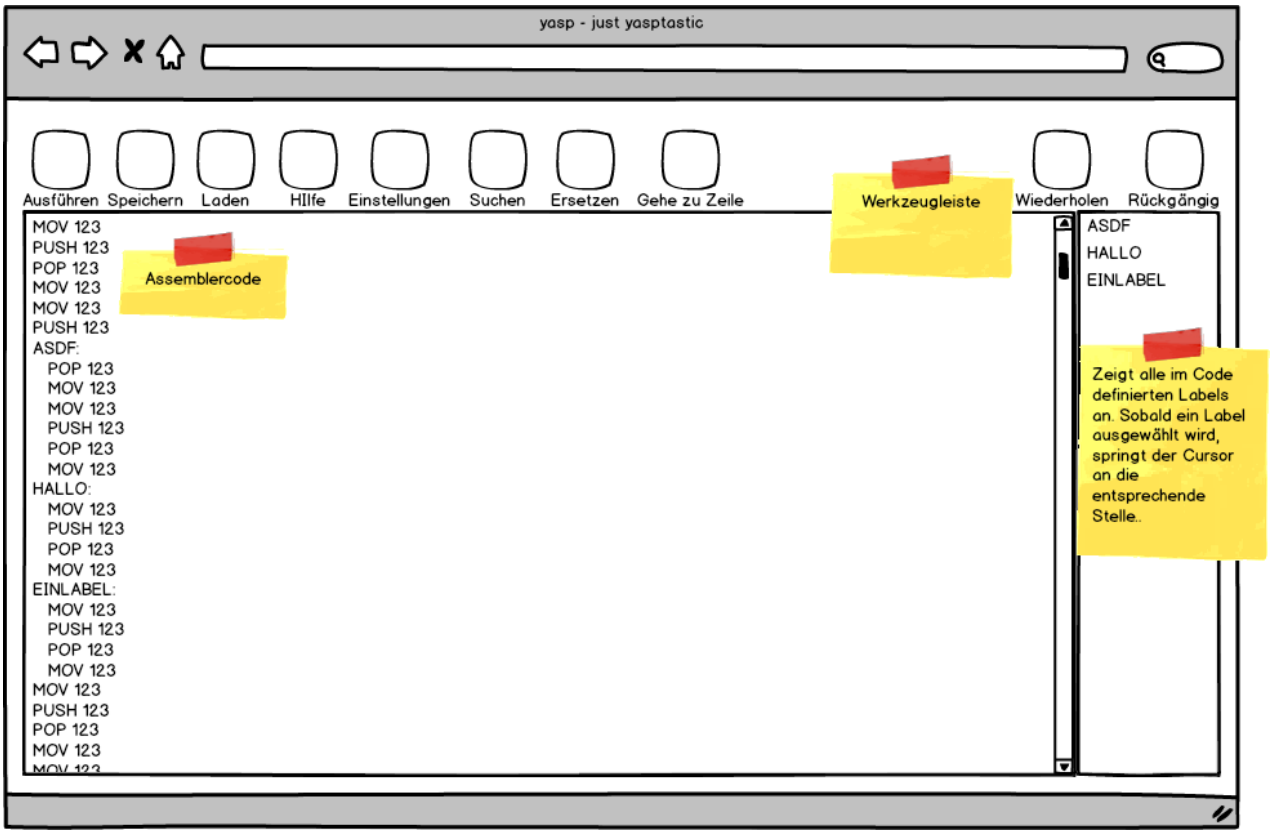
Registers
Die linke Seite zeigt den aktuellen Inhalt des Stacks an. Zusätzlich bezeichnet `SP` den Inhalt des Stackpointers an. Das `Carry Bit` und `Zero Bit` zeigen den jeweiligen Inhalt der Flags an. Alle Werte (Stackpointer, Carry Bit und Zero Bit) können vom Anwender bearbeitet werden.

Breakpoints
Die Tabelle zeigt die jeweiligen Breakpoints an, die im Quellcode definiert wurden. Position ist die Zeile des Breakpoints. Condition ist die Bedingung bei der, der Debugger anhält. Nur wenn jener Ausdruck wahr ist, wird dort angehalten.
Active gibt an ob beim Breakpoint pausiert wird oder ob er vorübergehend deaktiviert ist.
Jump To springt zur jeweiligen Zeile im Quellcode.

Assembler
Der Assembler an sich besitzt kein Userinterface und wird vollständig im Hintergrund ausgeführt. Er wird gestartet wenn der User auf den *Ausführen*-Button klickt.

Emulator
Der Debugger ist das Frontend des Emulators. Weshalb der Emulator selbst im Hintergrund läuft. Gestartet wird der Emulator durch den Klick auf *Ausführen*.

IDE



(Dieser Screen ist lediglich ein Mockup und kann sich im Laufe der Entwicklung ändern)

Qualitätsanforderungen

Siehe Anhang 3, Abnahmeprotokoll.

Nichtfunktionale Anforderungen

Datenschutz

Da das Projekt *yasp* eine Serverkomponente hat und dementsprechend Daten der Schülerinnen und Schüler gespeichert werden, muss sichergestellt sein, dass keine Daten in falsche Hände geraten. Dies wird durch optional setzbare Passwörter erreicht.

Anhänge

1. Glossar
2. Befehlsdokumentation
3. Abnahmeprotokoll
4. Protokolle und Datenstrukturen
5. Hardware
6. Emulator Aktivitätsdiagramme

Anhang 1: Glossar

Begriff	Erklärung
Emulator	Führt Maschinencode auf Plattformen aus, welche ganz andere Umgebungsbedingungen bereitstellen und erfordern. Dabei werden auf der auszuführenden Plattform die entsprechenden Komponenten der jeweiligen Zielplattform implementiert.
Assembler	Übersetzt Quellcode in Maschinencode.
IDE	Steht für "Integrated Development Environment". Eine möglichst komfortable Umgebung, die so abgeschlossen (d.h. ohne externe Tools) wie möglich agiert.
Entwicklungsumgebung	Siehe IDE.
Debugger	Erlaubt das genaue Inspizieren von laufenden Programmen und den Werten im Speicher.
Programmiersprache	Eine Programmiersprache definiert eine Syntax, die dann vom Computer als Logikoperationen ausgeführt werden kann.
CPU	„Gehirn des Computers" - führt die Befehle aus.
Maschinencode	Wird direkt von der CPU ausgelesen und ausgeführt.
Dialekt (Programmiersprachen)	Eine Abwandlung einer anderen Programmiersprache. Analog zu einem Dialekt in menschlichen Sprachen.
Editor	Ein Feld, in das man Text (oft Quelltext) eingeben kann.
Syntaxhighlighting	Hebt gewisse Schlüsselwörter im Editor hervor.
Quellcode, Quelltext oder Sourcecode	Mehrere Befehle welche in einer bestimmten Reihenfolge geschrieben werden um ein Ziel zu erreichen.
localStorage	Ca. ~5MB Speicher, die jeder Website zur Verfügung stehen.
Bitcode	Siehe Maschinencode
RAM	Random Access Memory: Ein flüchtiger (= temporärer) Speicher in dem Programmdaten gespeichert werden können.

JavaScript	Eine dynamisch typisierte Programmiersprache, die auf allen Browserplattformen zur Verfügung steht.
ECMA Script	Siehe JavaScript
Breakpoint	Wenn die Programmausführung einen Breakpoint erreicht, wird pausiert und man kann alle notwendigen Informationen im Debugger begutachten.
Web Worker	Ermöglicht die parallele Ausführung von Befehlen in Browsern.
Callback	Eine Funktion, die aufgerufen wird, sofern ein Ereignis bzw. ein Ergebnis einer Berechnung vorliegt.
PC (Program Counter)	Ein Wert im Emulator, der auf den aktuell auszuführenden Befehl zeigt.
Mockup / Wireframe	Eine nicht finale Darstellung einer Softwarekomponente. Dient hauptsächlich um die generelle Funktionsweise näherzubringen.
Build (Nomen)	Eine lauffähige Version eines Programms.
Potentiometer	Ein variabler Widerstand
Bitshift	Verschieben von Bits (001 => 010)
Nullterminiert	Das letzte Byte eines nullterminierten Strings ist \0, wodurch erkannt werden kann, wo ein String aufhört.
Alias	Kosename
Rekursion	Siehe Rekursion.

Anhang 2: Befehlsdokumentation

Ein Befehl

Jeder Befehl befindet sich in einer eigenen JavaScript-Datei. In dieser Datei befinden sich Metadaten zum Befehl, sowie der eigentliche JavaScript-Code um den Befehl auszuführen. Diese Informationen werden vom Parser, Emulator und außerdem zum Erzeugen der Dokumentation verwendet.

Befehlsformat

```
{
  "name": "MOV",
  "description": "",
  "code": [
    {
      "value": [ string | number ],
      "length": 8
    }
  ],
  "params": [
    {
      "type": [ "r_byte", "r_word",
                "l_byte", "l_word",
                "pin", "address"   ]
    }
  ],
  "exec": function () {}
}
```

Code

Die Code-Eigenschaft gibt den statischen Start eines Befehls an. Dieser wird vom Emulator verwendet um den Befehl zu erkennen.

- `value`, der Wert nach dem gesucht werden soll.
  - als String: binär
  - als Zahl: literal
- `length`, Länge des Wertes in Bits. Darf 8 nicht übersteigen.

Parameter

Die Parameter werden nach dem letzten **Bit** des `Code` gelesen.

Parametertypen

- `r_byte`
  - Länge: 5 Bits
  - Nummer eines Byte-Registers (RAM Adresse: `value`)

- `r_word`
  - Länge: 5 Bits
  - Nummer eines Word-Registers (RAM Adresse: `value * 2`)
- `l_byte`
  - Länge: 8 Bits
  - Byte literal
- `l_word`
  - Länge: 16 Bits
  - World literal
- `pin`
  - Länge: 5 Bits
  - Nummer eines Pins
- `address`
  - Länge: 11 Bits
  - ROM-Adresse, z.B. für Sprunginstruktionen

`Exec`-Funktion

Die `exec`-Funktion wird im Kontext des Emulators aufgerufen und hat somit vollständigen Zugriff auf alle Funktionen und Speicherwerte.

Parameter

Vor dem Ausführen der Funktion werden alle benötigten Werte geladen und als Parameter zur Verfügung gestellt:

```
{
  "value": [ null | number ],
  "address": [ null | number ]
}
```

- `r_byte`, `r_word`
  - `value`: Wert des Registers
  - `address`: Adresse des Registers im RAM
- `l_byte`, `l_word`
  - `value`: Angegebener Wert
  - `address`: null
- `pin`
  - `value`: aktueller Zustand des Pins
  - `address`: Nummer des Pins
- `address`
  - `value`: Adresse im RAM
  - `address`: null

Emulator-Architektur

Der Emulator muss in der Lage sein die Assembler Befehle auszuführen. Dabei wird jeder Befehl in die folgenden Befehle abstrahiert, um ein flexibles und erweiterbares Design zu ermöglichen.

- Register
  - `WriteWordRegister(n, v)`
  - `WriteByteRegister(n, v)`
  - `ReadWordRegister()`
  - `ReadByteRegister()`

- Stack
  - `PushWord(v)`
  - `PushByte(v)`
  - `PopWord()`
  - `PopByte()`
- RAM / ROM
  - `ReadROM(o)`
  - `WriteROM(o, v)`
  - `ReadRAM(o)`
  - `WriteRAM(o, v)`
- PC / Flags
  - `WritePC(pc)`
  - `ReadPC()`
  - `ReadFlags()`
  - `WriteFlags(flags)`
  - `isCarryFlagSet()`
  - `isZeroFlagSet()`
- Debugging
  - `Debug()`
  - `Echo()`
  - `Break()`
  - `Error()`
- Hardware
  - `SetIO()`
  - `GetIO()`
- `Wait()`

Instruktionen

Register-Befehle

- `MOV`
- `ADD`
- `SUB`
- `INC`
- `DEC`
- `INV`
- `CLR`
- `RR`
- `RL`
- `CMP`
- `AND`
- `OR`
- `XOR`
- `MUL`
- `DIV`

Stack

- `PUSH`
- `POP`

Sprungbefehle

- `JMP`

- GOTO
  alias für JMP.
- CALL
- RETURN
- JZ
- JNZ
- JC
- JNC
- JMPI

Interrupts
- ENABLE
- DISABLE
- RETI
- LA
  kinda-alias für MOV.

Debugging
- DEBUG
- BREAK
- ECHO

Hardwarebefehle
- HIGH
- LOW
- TOGGLE
- PIN
- POT
- ADC1

Sonstiges
- DELAY
- PAUSE
  alias für DELAY.
- RANDOM

Zusätzliche Befehle
Diese Befehle sind nicht in der Dokumentation definiert, sind aber für Folgeprojekte sinnvoll und werden deshalb zusätzlich implementiert.
- WRITERAM
- READRAM

Direktiven
- ORG
  Setzt die aktuelle Position im Bitcode.
- STRING
  Schreibt einen nullterminierten String in den Bitcode.
- DA
  Schreibt die Adresse des gegebenen Labels in den Maschinencode.
- DB

Schreibt ein Byte in den Bitcode.

- `END`
  Beendet die Assemblierung.
- `DEFINE`
  Definiert einen Alias.

Anhang 3: Abnahmeprotokoll

Kategorie	Testfall	KO	OK?	Kommentar
Emulator/Assembler	Testprogramm 01	Ja		
Emulator/Assembler	Testprogramm 02	Ja		
Emulator/Assembler	Testprogramm 03	Ja		
Emulator/Assembler	Testprogramm 04	Ja		
Emulator/Assembler	Testprogramm 05	Ja		
Emulator/Assembler	Testprogramm 06	Ja		
Emulator/Assembler	Testprogramm 07	Ja		
Emulator/Assembler	Testprogramm 08	Ja		
Emulator/Assembler	Testprogramm 09	Ja		
Emulator/Assembler	Testprogramm 10	Ja		
Emulator	Unit Tests	Ja		
Assembler	Unit Tests	Ja		
IDE	Öffnen	Ja		
IDE	Schließen	Ja		
IDE	Einstellungen verändern	Nein		
IDE	Quelltext eingeben	Ja		
IDE	Suchen	Nein		
IDE	Ersetzen	Nein		
IDE	Gehe zu Zeile	Nein		
IDE	Schnellhilfe	Nein		
IDE	Hilfe	Nein		
IDE	Speichern	Ja		

IDE	Laden	Ja		
IDE	Rückgängig	Ja		
IDE	Wiederholen	Ja		
Debugger	Break	Ja		
Debugger	Step	Ja		
Debugger	Step Back	Nein		
Debugger	Continue	Ja		
Debugger	PC setzen	Nein		
Debugger	Execute	Nein		
Debugger	Breakpoint setzen	Ja		
Debugger	Breakpoint entfernen	Ja		
Debugger	Breakpoint temporär deaktivieren	Nein		
Debugger	Bedingte Breakpoints setzen	Nein		
Debugger	Zeilenbezogene Bedingte Breakpoints	Nein		

Gesamtprodukt abgenommen:

Ja	Nein

(Zutreffendes Ankreuzen)

_____	_____
Robert FISCHER	Michael LUTONSKY
Projektmitglied	Projektleiter
_____	_____
Dipl.- Ing. Robert JELINEK	Dipl.- Ing. Joachim GRÜNEIS
Auftraggeber/Begleitlehrer	Begleitlehrer

Wien, am _____

Testfälle

Testprogramme

Die Testprogramme befinden sich im `doc/sampleprograms/`-Verzeichnis.

01. Die eingeschaltene LED vom Potentiometer abhängig

Testdaten	Der Potentiometer wird gedreht
Testresultat	1. Drittel: Rote LED leuchtet 2. Drittel: Gelbe LED leuchtet 3. Drittel: Grüne LED leuchtet
Ausführungsmodus	Run

02. Pulsweitenmodulation

Testdaten	Buttons einzeln gedrückt halten.
Testresultat	Die Rote LED wird heller und dunkler.
Ausführungsmodus	Run

03. Pulsweitenmodulation vom Potentiometer abhängig

Testdaten	Der Potentiometer wird gedreht
Testresultat	Die Rote LED pulsiert in der eingestellten Geschwindigkeit
Ausführungsmodus	Run

04. Grundlegender Funktionstest

Testdaten	Debug und Register auf den DEC-Modus schalten.

Testresultat	Step 1: Register b0 hat den Wert 23 Step 2: Register b1 hat den Wert 67 Step 3: Register b0 hat den Wert 67 Step 4: Register b0 hat den Wert 68 Step 5: Register b0 hat den Wert 135 Step 6: Der Wert 135 im Debug-Fenster Step 7: Zurück zu Step 1
Ausführungsmodus	Debug

05. LEDs werden via Interrupts ein- und ausgeschalten

Testdaten	Taster 1 drücken Taster 2 drücken
Testresultat	Die grüne LED blinkt. Grauer Taster gedrückt: Rote LED wird umgeschalten Roter Taster gedrückt: Gelbe LED wird umgeschalten
Ausführungsmodus	Run

06. LEDs werden via Polling ein- und ausgeschalten

Testdaten	Grauen Taster drücken.
Testresultat	Rote LED leuchtet nur wenn der Taster nicht gedrückt ist.
Ausführungsmodus	Run

07. Eine Variable inkrementieren wenn ein Button gedrückt wird

Testdaten	Taster 1 oder 2 drücken
Testresultat	Taster 1 gedrückt: ausgegebener Wert erhöht sich Taster 2 gedrückt: ausgegebener Wert verringert sich
Ausführungsmodus	Run

08. LED blinken

Testdaten	Keine
Testresultat	Rote LED blinkt
Ausführungsmodus	Run

09. 3 LEDs werden als Ampel mit fixem Zeitintervall geschaltet

Testdaten	Keine
Testresultat	Rote LED leuchtet Rote und gelbe LED leuchtet Grüne LED leuchtet Grüne LED blinkt vier mal Keine LED leuchtet Gelbe LED leuchtet => Zum Anfang
Ausführungsmodus	Run

10. Bitshift

Testdaten	Debug und Register auf den DEC-Modus schalten.
Testresultat	Step 1: Register b2 hat den Wert 1 Step 2: Register b3 hat den Wert 128 Step 3: Register b1 hat den Wert 7 Step 4: Register b0 hat den Wert 18 Step 5: Register b0 hat den Wert 19 Step 6: Register b0 hat den Wert 147 Step 7: Register b2 hat den Wert 2 Step 8: Register b3 hat den Wert 64 Step 9: Register b1 hat den Wert 6 Step 10: Step 4 bis 9: 7 mal ausführen Step 11: von vorne
Ausführungsmodus	Debug

Unit-Tests Assembler

Testdaten	Alle während der Entwicklung geschriebenen Unit-Tests werden verwendet.
Testresultat	Erfolgreiche Ausführung

Unit-Tests Emulator

Testdaten	Alle während der Entwicklung geschriebenen Unit-Tests werden verwendet.
Testresultat	Erfolgreiche Ausführung

IDE

Öffnen

Testdaten	Die IDE wird geöffnet, indem man die entsprechende URL eingibt
Testresultat	Die IDE ist offen, alle relevanten Komponenten sind einsatzbereit und es ist zu keinem Absturz gekommen.

Schließen

Testdaten	Die IDE wird geschlossen, indem man das entsprechende Browser Fenster bzw. Tab schließt.
Testresultat	Die IDE ist nicht mehr aktiv und alle relevanten Daten sind im `localStorage` gespeichert. • Einstellungen • Quelltext (falls gespeichert)

Einstellungen verändern

Testdaten	Einstellungen Dialog öffnen und folgende Einstellungen verändern: • Syntax Highlighting Theme ändern • Leerzeichen eines Tabs (auf 4 ändern) • Automatischen Speichern abschalten • Autovervollständigung deaktivieren (falls implementiert) • Sprache auf Englisch stellen • Die Label-Liste immer anzeigen • Die Quickhelp immer anzeigen
Testresultat	Die Einstellungen wurden geändert und sind nun aktiv.

Quelltext eingeben

Testdaten	Testfall 01 in die IDE eingeben.
Testresultat	Das Eingabefenster der IDE beinhaltet den eingegebenen Quelltext.

Suchen

Testdaten	1. Testfall 01 in die IDE laden. 2. Öffnen der Suchfunktion via Shortcut oder Menüpunkt 3. Eingabe des Suchterms *JMP*.
Testresultat	Das erste Suchergebnis von *JMP* wird hervorgehoben und das Eingabefenster springt an die Position.

Ersetzen

Testdaten	1. Testfall 01 in die IDE laden. 2. Öffnen der Ersetzen-Funktion via Shortcut oder Menüpunkt 3. Eingabe des Suchterms *JMP*. 4. Eingabe des Ersetzungsterms *GOTO*. 5. Jedes Vorkommnis mit *Yes* Bestätigen.
Testresultat	Jedes Vorkommnis von *JMP* ist nun ein *GOTO*.

Gehe Zu Zeile

Testdaten	1. Testfall 01 in die IDE laden. 2. Öffnen des entsprechenden Dialogs im Menü 3. Eingabe der gewünschten Sprungzeile: 42.
Testresultat	Die Zeile 42 wird hervorgehoben und das Eingabefenster springt an die entsprechende Position.

Schnellhilfe

Testdaten	1. Testfall 01 in die IDE laden. 2. Cursor zu einem bestimmten Befehl bewegen.
Testresultat	Es werden die Befehlsdefinition, die Beschreibung und die entsprechenden Parameter angezeigt.

Hilfe

Testdaten	1. Hilfebutton mit der Maus drücken 2. Nach den Befehl *MOV* suchen
Testresultat	Es werden die 4 Überladungen des *MOV*-Befehles in dem Dialog sichtbar.

Speichern

Testdaten	1. Testfall 01 in die IDE laden. 2. Beliebigen Befehl ändern. 3. Lokal Speichern (via Menüpunkt oder Shortcut)
Testresultat	Der Quellcode wird im `localStorage` gespeichert (MUSS). Weiters wird der Quellcode, falls gewollt, am Schulserver gespeichert (KANN).

Laden

Testdaten	1. Öffnen (via Menüpunkt oder Shortcut). 2. Zuvor gespeicherte Datei öffnen
Testresultat	Der zuvor gespeicherte Quellcode wurde geladen und wird in der IDE angezeigt.

Rückgängig

Testdaten	1. Testfall 01 in die IDE laden. 2. Code eingeben (z. B.: Kommentar hinzufügen). 3. Rückgängigfunktion aufrufen (via Menüpunkt oder Shortcut)
Testresultat	Der ursprüngliche Quellcode ist im Eingabefenster.

Wiederholen

Testdaten	1. Testfall 01 in die IDE Laden. 2. Code eingeben (z. B.: Kommentar hinzufügen). 3. Rückgängigfunktion aufrufen (via Menüpunkt oder Shortcut). 4. Wiederholenfunktion aufrufen (via Menüpunkt oder Shortcut)
Testresultat	Der ursprüngliche Quellcode mit den Änderungen ist sichtbar.

Debugger

Break

Testdaten	1. Testfall 01 in die IDE laden. 2. Ausführung starten. 3. Break Funktion ausführen.
Testresultat	Pausiert die Ausführung des Programms am momentanen PC. Die Daten können im Debugger begutachtet werden.

Step

Testdaten	1. Testfall 01 in die IDE laden. 2. Ausführung starten. 3. Ausführung pausieren (mit Break-Funktion) 4. Step Funktion ausführen.
Testresultat	Der Programcounter wird inkrementiert, der nächste Befehl wird ausgeführt, der Emulator pausiert und man kann die Daten im Debugger begutachten.

Step Back

Testdaten	1. Testfall 01 in die IDE laden. 2. Ausführung starten. 3. Ausführung pausieren (mit Break-Funktion). 4. Step Funktion ausführen. 5. Step Back Funktion ausführen.
Testresultat	Der Zustand des Emulators ist der selbe, wie vor dem Ausführen von Step . Die Daten können im Debugger begutachtet werden.

Continue

Testdaten	1. Testfall 01 in die IDE laden. 2. Ausführung starten. 3. Ausführung pausieren (Break). 4. Ausführung fortfahren (Continue)
Testresultat	Das Programm läuft normal weiter.

PC setzen

Testdaten	1. Testfall 01 in die IDE laden. 2. Ausführung starten. 3. Auf "Jump to" clicken 4. auf eine beliebige Zeile klicken.
Testresultat	Der PC wird gesetzt und der Emulator gestoppt. Die Ausführung kann mit "Continue" gestartet werden.

Execute

Testdaten	1. Testfall 01 in die IDE laden. 2. Ausführung starten. 3. Assembler Befehl(e) 4. in die Execute Konsole eingeben.
Testresultat	Der eingegebene Quellcode wird ausgeführt und danach wird vom alten PC aus weiter ausgeführt.

Breakpoint setzen

Testdaten	1. Testfall 01 in die IDE laden. 2. Klick in der Breakpoint-Leiste in der entsprechenden Zeile
Testresultat	Ein Punkt erscheint und es wird pausiert sobald die Ausführung zur entsprechenden Position gelangt.

Breakpoint entfernen

Testdaten	1. Testfall 01 in die IDE laden. 2. Nachdem man einen Breakpoint erstellt hat, nochmal an die selbe Stelle klicken.
Testresultat	Der Breakpoint ist entfernt und es wird nicht mehr an der entsprechenden Position pausiert.

Breakpoint temporär deaktivieren

Testdaten	1. Testfall 01 in die IDE laden. 2. Breakpoint erstellen. 3. Breakpoint in der Breakpoint-Liste deaktivieren
Testresultat	Die Ausführung sollte nicht mehr angehalten werden, es bleibt aber weiterhin in der IDE ersichtlich, dass dort ein Breakpoint ist, der leicht wieder aktiviert werden kann.

Bedingte Breakpoints setzen

Testdaten	1. Testfall 01 in die IDE Laden. 2. Den Menüpunkt *Globale Breakpoints* auswählen.
Testresultat	Ein Dialog öffnet und man kann die entsprechende Bedingung eingeben. Die Ausführung pausiert, sobald die Bedingung erfüllt ist.

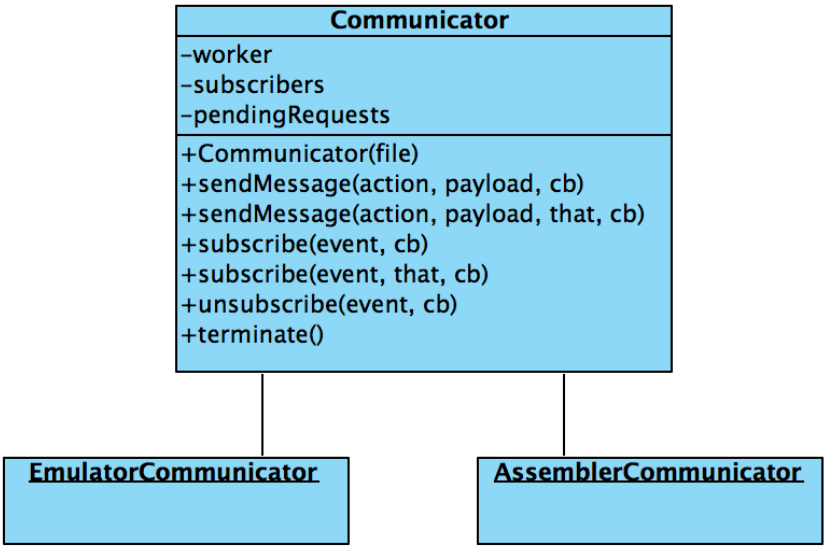
Zeilenbezogene bedingte Breakpoints setzen

Testdaten	1. Testfall 01 in die IDE laden. 2. Breakpoint erstellen. 3. Rechtsklick auf den Breakpoint und *Bedingter Breakpoint* auswählen.
Testresultat	Ein Dialog öffnet und man kann die entsprechende Bedingung festlegen. Wenn der Quellcode diese Zeile erreicht und die Bedingung erfüllt ist, wird pausiert.

Anhang 4: Protokolle und Datenstrukturen

In diesem Dokument werden die von *yasp* verwendeten Protokolle und Datenstrukturen näher beschrieben.

Communicator



Die Kommunikation mit den Workern findet über die `Communicator` Klasse statt. Diese kapselt den Webworker so weit wie möglich ab, so dass es einfacher ist Browserunabhängig zu bleiben und eine einheitliche Schnittstelle für andere Programmteile zu bieten.

Alle `cb`-Parameter haben folgende Signatur: `function (error, payload)` und den Rückgabewert `void`.

sendMessage(action, payload, cb)

Dabei kann mit der Methode `sendMessage` eine Nachricht an den jeweiligen Worker gesendet werden. Der Parameter `payload` enthält Informationen für den Worker. Wenn der Worker mit der Ausführung der Anweisung fertig ist, wird die `cb` Funktion aufgerufen.

subscribe(event, cb) / unsubscribe(event, cb)

Weiters werden für sog. *Broadcast Messages* (z.B.: Emulator hat die Ausführung angehalten) alle Callbacks aufgerufen die via `subscribe` angegeben wurden. Dabei ist `event` die Art der Broadcast Nachricht und `cb` die auszuführende Funktion.

terminate()

Beendet den Worker.

Webworker Protokoll

Jeder Web Worker kommuniziert mit Hilfe des hier definierten Protokolls. Die eigentliche Kommunikation mit dem Worker findet in der `Connector`-Klasse statt.

Request (s. communicator.sendMessage)

```
{
  "action": "",
  "id": "",
  "payload": { }
}
```

- `action`: was der Worker machen soll
- `id`: eindeutiger Bezeichner für diese Nachricht
- `payload`: Parameter für den Worker

Response (s. cb in communicator.sendMessage & communicator.subscribe)

```
{
  "action": "",
  "id": "",
  "error": { },
  "payload": { }
}
```

- `action`: aus dem **Request** oder name des Broadcasts
- `id`: aus dem **Request** oder `null` bei Broadcast
- `payload`: Rückgabe des Workers
- `error`: Fehlermeldungen des Workers

Assembler

Actions

Assemblieren - ASSEMBLE

Request-Payload

```
{
  "code": "",
  "jobs": [ "symbols", "bitcode", "map" ]
}
```

Response-Payload

```
{
  "bitcode": new Uint8Array(),
  "symbols": { }
  "map": { }
}
```

- `bitcode`: Maschinencode der v om Emulator ausgeführt werden kann
- `symbols`: siehe Datenstrukturen/Symboltabelle
- `map`: siehe Datenstrukturen/Map

Response-Error (oder `null`)

```
{
  "errors": [ ]
}
```

- `errors`: siehe Datenstrukturen/Assemblierungsfehler

Datenstrukturen

Symboltabelle

```
{
  "labels": [ ],
   "usedRegisters": [ ],
   "defines": [ ],
   "instructions": [
     "MOV": 0
   ]
}
```

- `labels`: alle Labelnamen in alphabetischer Reihenfolge
- `usedRegisters`: alle **verwendeten** Register geordnet nach Häufigkeit

- `defines`: alle Namen aller `#defines` in alphabetischer Reihenfolge
- `instructions`: alle verwendeten Instruktionen und deren Häufigkeit

Map

Die Map macht es möglich den Befehl zu einer Zeilennummer zu finden. Sie wird als Object dargestellt und enthält als Key die Zeilennummer und als Value das entsprechende Offset im zurückgegebenen Maschinencode. Es ist möglich, dass eine Zeile keinen entsprechenden OpCode hat (zum Beispiel Kommentare).

```
{
  0: 0,
  1: 3,
  2: 4,
  4: 8
}
```

Assemblierungsfehler

```
{
  "success": [ true | false ],
  "errors": [
    {
      "type": [ "error" | "warning" ],
      "name": "E_ERR",
      "line": 1,
      "char": 3,
      "message": ""
    }
  ]
}
```

Broadcasts

Keine.

Emulator

Actions

Lädt den Code in den Speicher - `LOAD`

Request-Payload

```
{
  "bitcode": new Uint8Array(),
  "start": 0
}
```

- `start` bezeichnet wo der neue Bitcode geladen werden soll. Hierbei wird nicht der gesamte Speicher zurückgesetzt sondern nur von `start` bis `start + bitcode.length` geschrieben. Möglicher Inhalt des Speichers wird dabei überschrieben.

Response-Payload

```
{ }
```

Error-Payload (oder `null`)

```
{
  "code": 0
}
```

- `code` ist der entsprechende Fehlercode. Dabei gibt es folgende mögliche Literale:

Code	Beschreibung
0	Start ungültig
1	Out of memory
2	Falscher Datentyp

Code von aktuellem PC ausführen - CONTINUE

Request-Payload

```
{
  "count" : [null | 0]
}
```

- `count` Anzahl der Instruktionen. Null bedeutet, dass es uneingeschränkt ausführt.

Response-Payload

```
{ }
```

Error-Payload (oder `null`)

```
{
  "code": 0
}
```

- `code` ist der entsprechende Fehlercode. Dabei gibt es folgende mögliche Literale:

Code	Beschreibung
0	Count ungültig (Negativ oder außerhalb des erlaubten Bereichs)
1	Falscher Datentyp

Ausführung anhalten - `BREAK`

Request-Payload

```
{ }
```

Response-Payload

```
{ }
```

Zustand des Emulators abfragen - `GET_STATE`

Request-Payload

```
{ }
```

Response-Payload

```
{
  "state": { }
}
```

Error-Payload

```
{
  "code": 0
}
```

Code	Beschreibung
0	Emulator läuft

- `state` beinhaltet das State Objekt des Emulators. Für weitere Informationen, siehe Datenstrukturen / State.

Zustand des Emulators setzen - `SET_STATE`

Request-Payload

```
{
  "state": { }
}
```

Response-Payload

```
{ }
```

Error-Payload

```
{
  "code": 0
}
```

Code	Beschreibung
0	Ungültiger ROM (zu groß/klein/...)

1	Ungültiger RAM (zu groß/klein/...)
2	Ungültige Register
3	Ungültiges IO

Setzt die Liste der Breakpoints - `SET_BREAKPOINTS`

Request-Payload

```
{
  "breakpoints": [{ }]
}
```

Response-Payload

```
{ }
```

Error-Payload

```
{
  "code": 0
}
```

Code	Beschreibung
0	Ungültiges Array
1	Ein Breakpoint Objekt ist ungültig und/oder eines der Attribute des Breakpoints.

Ausführung von beliebigem Bitcode - `EXECUTE`

Request-Payload

```
{
  "bitcode": new Uint8Array(),
}
```

Response-Payload

```
{ }
```

Error-Payload

```
{
  "code": 0
}
```

Code	Beschreibung
0	Ungültiges Array
1	Emulator nicht im `STOPPED` State

Broadcasts

IO verändert - `IO_CHANGED`

Broadcast-Payload

```
{
  "io" : { }
}
```

- `io` beinhaltet das IO Objekt, welches sich verändert hat.

Ausführung angehalten - `BREAK`

Broadcast-Payload

```
{
  "reason": [ "breakpoint" | "break" | "divide_by_zero" ]
}
```

Ausführung fortgefahren - `CONTINUED`

Broadcast-Payload

```
{
}
```

Code geladen - `LOADED`

Broadcast-Payload

```
{
  "start": 0,
  "length": 0
}
```

Debug Nachricht - `DEBUG`

Broadcast-Payload

```
{
  "message": "breakpoint",
  "register": {
    "type": "b",
    "number": "2",
```

```
    "value": 0
  }
}
```

- Entweder `message` oder `register` kann `null` sein.

Datenstrukturen

State

Ein State beschreibt den aktuellen Status des Emulators. Dabei enthält dies den aktuellen PC, RAM, ROM, Register und weitere Informationen. Der State wird dazu verwendet um im Debugger die entsprechenden Informationen (wie z.B.: Registerinhalte) anzuzeigen.

```
{
  "rom": new Uint8Array(),
  "ram", new Uint8Array(),
  "registers": {
    "general": {“b”: {}, “w”: {} },
    "special": {
      "pc": 0,
      "sp", 0
    },
    "flags": {
      "C", [true | false],
      "Z", [true | false],
    }
  },
  "io": [{ }]
}
```

- `rom` beinhaltet den ROM des Emulators als Byte Array
- `ram` beinhaltet den RAM des Emulators als Byte Array
- `registers` beinhaltet die diversen Registers des Emulators
  - `general` sind die 64 General Purpose Register
  - `special` sind die Special Purpose CPU Register
    - `pc` Program Counter
    - `sp` ist der Stack Pointer
  - `flags` die Flags des Emulators
    - `C` Carry Bit
    - `Z` Zero Bit
- `io` beinhaltet alle IO Objekte (siehe IO für weitere Informationen) des Emulators als Array.

IO

Das sind Objekte die den Zustand eines Input/Output Objektes bezeichnen. IO Objekte können mit LED, Taster, Potentiometer, Schrittmotoren oder Pins verbunden sein.

```
{
  "pin": 0,
  "type": ["gpio" | "adc"],
  "mode": ["in" | "out"],
  "state": [0 | true | false]
}
```

- `type` bezeichnet welches IO Objekt gemeint ist.

Breakpoint

Ein Breakpoint zeigt auf einen Punkt im Code an dem die aktuelle Programmausführung gestoppt werden soll.

```
{
  "offset": [null | 0],
  "condition": [null | { }]
}
```

- `offset` bezeichnet den Opcode an dem der Breakpoint gesetzt wurde. Wenn er `null` ist, bedeutet dies, dass angehalten wird, sobald die `condition` erfüllt ist, was bedeutet, dass der Emulator diesen Breakpoint nach jedem Ausführen eines Opcodes überprüfen muss.
- `condition` ist die Bedingung, bei der gestoppt werden soll. Die `condition` ist ein Condition Objekt, für weitere Informationen siehe Datenstrukturen / Condition. Wenn `condition` `null` ist, darf `offset` nicht `null` sein und vice versa.

Condition

Eine Condition bezeichnet eine Bedingung, die verwendet wird um zu bestimmen ob bei einem Breakpoint gestoppt werden soll. Dabei wird der Bezugswert genommen (errechnet sich aus `type` und `param`) und wird mit dem Wert in `value` mit dem Vergleichsoperator `operator` verglichen.

```
{
  "type": ["register" | "flag" | "io" | "ram" | "rom"],
  "param": [0 | ""],
  "operator": ["=", "!=", "<", ">", "<=", ">=", "write"],
  "value": [new Uint8Array() | 0]
}
```

- `type` bezeichnet was die Bedingung überprüft.
- `param` ist der zusätzliche Parameter, der näher spezifiziert was genau untersucht werden soll.

Typ	Bedeutung des Parameters	Möglichkeiten
register	Welches Register ist gemeint	W0-31 (Wordregister) B0-31 (Byteregister) PC (Program Counter) SP (Stack Pointer)
flag	Welches Flag ist gemeint	C (Carry Bit) Z (Zero bit)
io	Welcher Pin ist gemeint	0 - 15
RAM / ROM	Offset von dem angefangen wird zu vergleichen.	0,1,2,3,...

- `operator` besagt welche Operation zwischen dem Bezugswert (der sich aus der Kombination von `type` und `param` ergibt) angewandt werden soll.
- `value` bezeichnet den Zielwert, bei dem angehalten wird. Je nach `type` und `param` müssen andere Datentypen in `value` sein:

Typ	Parameter	Datentyp
register	W0-31	Word
register	B0-31	Byte
register	PC / SP	Integer
register	C / Z	Boolean
io	0-15	Zahl
RAM / ROM		Das Bytearray das ab dem angegebenen Offset verglichen wird.

Server

Der Server welcher den Sourcecode der Nutzer speichert verfügt über eine simple REST-API. Ein Schüler oder eine Schülerin darf nur seine eigenen Dateien abrufen. Ein Lehrer oder eine Lehrerin darf alle Dateien abfragen. Die Authentifizierung wird über den ActiveDirectory-Server der Schule gelöst. Die Übergabe des Usernames und des Passwords erfolgt über basic-authentication.

GET /file.php

Gibt eine Liste aller Dateien zurück.

Format

```
[
  {
    "filename": "",
    "username": "",
    "createdate": "",
    "group": "",
    "changedate": ""
  }
]
```

GET /file.php?username=USER

Gibt eine Liste aller Dateien des angegebenen Users zurück.

Format

Siehe `GET /file`.

GET /file.php?username=USER&filename=FILE

Gibt den Inhalt einer Datei zurück.

Format

```
{
  "filename": "",
  "username": "",
  "createdate": "",
  "changedate": "",
  "group": "",
  "content": ""
}
```

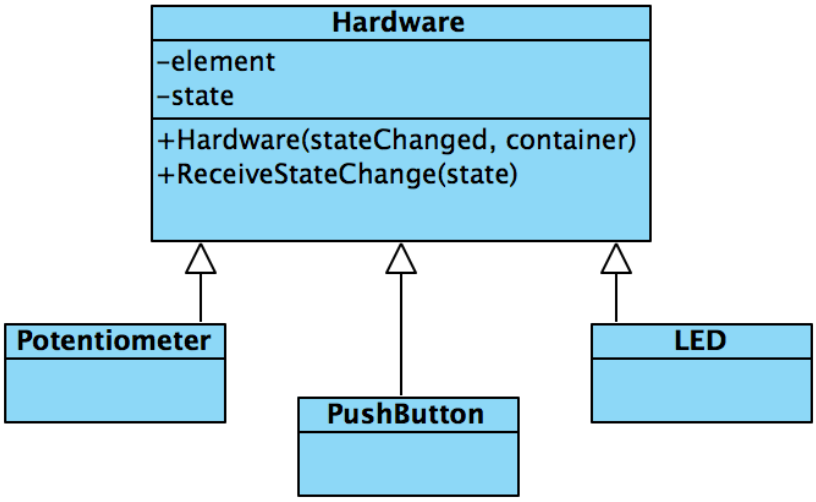
POST /file.php?username=USER&filename=FILE

Speichert oder aktualisiert die Datei des Users mit dem angegebenen Namen. Im Requestbody befindet sich der Inhalt der Datei.

DELETE /file?username=USER&filename=FILE

Löscht die angegebene Datei.

Anhang 5: Hardware



Beschreibung

Die `Hardware` Klasse beschreibt die einzelnen Hardwarekomponenten des USB Masters. Dabei wird die konkrete Hardware abstrahiert und ein generisches Zugriffsinterface angeboten. Durch diese Abstrahierung wird es zukünftig möglich sein, neue Hardwarelemente leicht zu implementieren.

Methoden

Hardware(stateChanged, container)

Ist der Konstruktor und entnimmt ein Callback `stateChanged` entgegen, das aufgerufen wird, falls sich der Status der Hardware ändert und zusätzlich nimmt der Konstruktor den Parameter `container` entgegen der das HTML Element festlegt, das diese Hardware repräsentiert. Die Implementierung der Zustandsänderungen des HTML Elements werden in der Implementierung der `Hardware` Klasse festgelegt sein.

ReceiveStateChange (state)

Wird aufgerufen wenn sich der Status des Pins mit dem die Hardware verbunden ist. Zum Beispiel durch die Ausführung eines `HIGH` oder `LOW` Befehls. Das Hardwareelement muss auf die Änderung reagieren und zum Beispiel die LED ein- oder ausschalten.

Hardwaretypen

LED

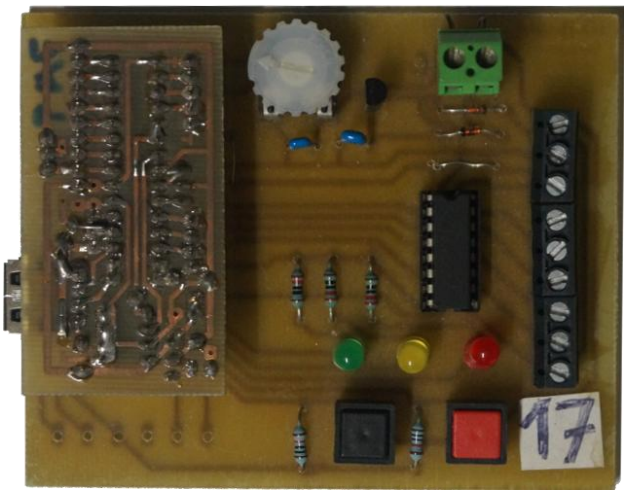
Pin-In: Binär
Pin-Out: Nein
Erscheinung: Ein / Aus

Taster

Pin-In: Keine
Pin-Out: Binär, standard: `OFF`, solange der User den Taster gedrückt hält: `ON`
Erscheinung: Gedrückt / nicht gedrückt

Potentiometer
Pin-In: Nein
Pin-Out: Analog
Erscheinung: Drehregler

Hardwareeinsatz



LED

- Rot
  - an Pin 5, type gpio, mode out
- Gelb
  - an Pin 4, type gpio, mode out
- Grün
  - an Pin 3, type gpio, mode out

Taster

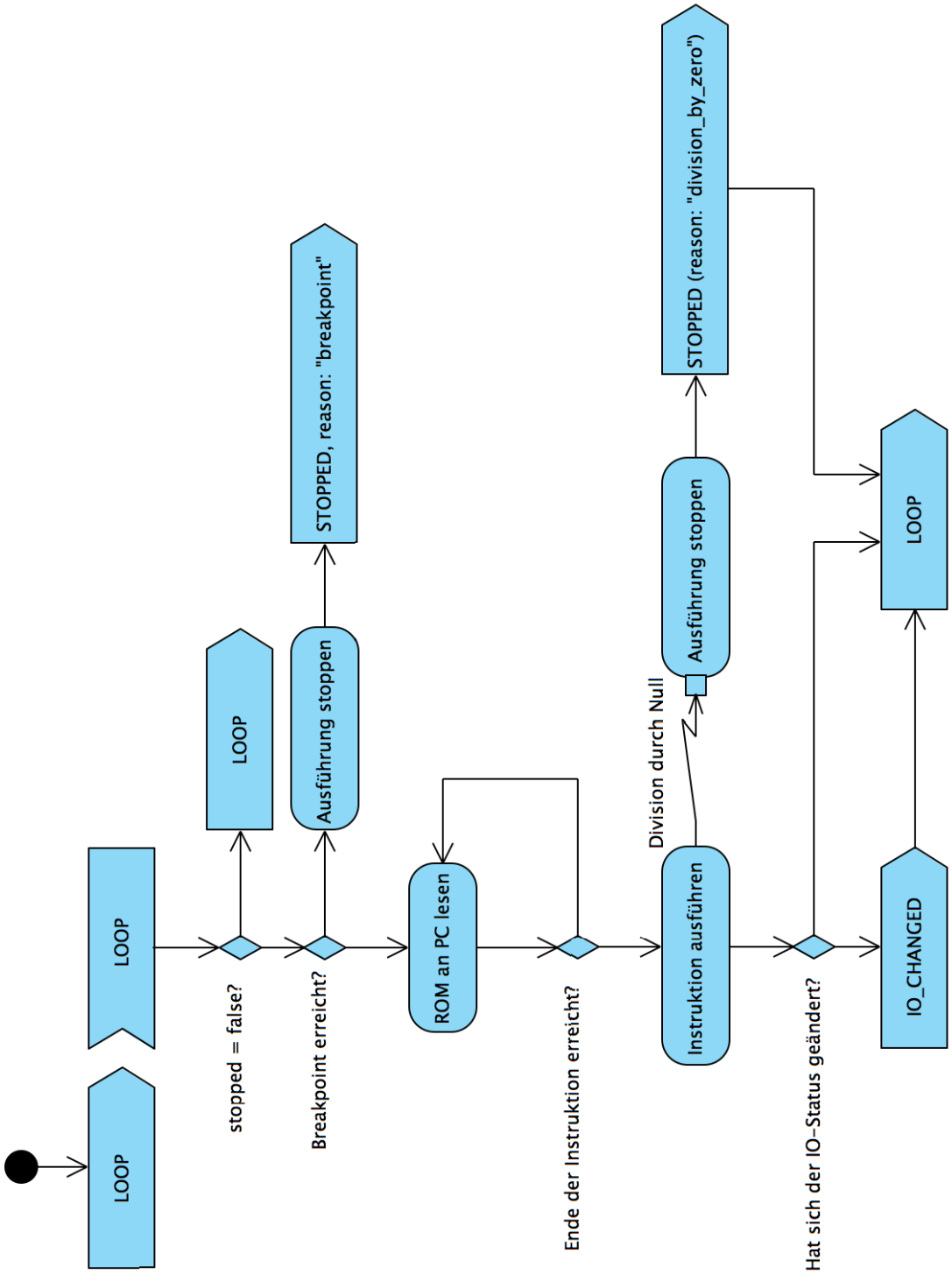
- Rot
  - an Pin 1, type gpio, mode in
- Schwarz
  - an Pin 2, type gpio, mode in

Potentiometer

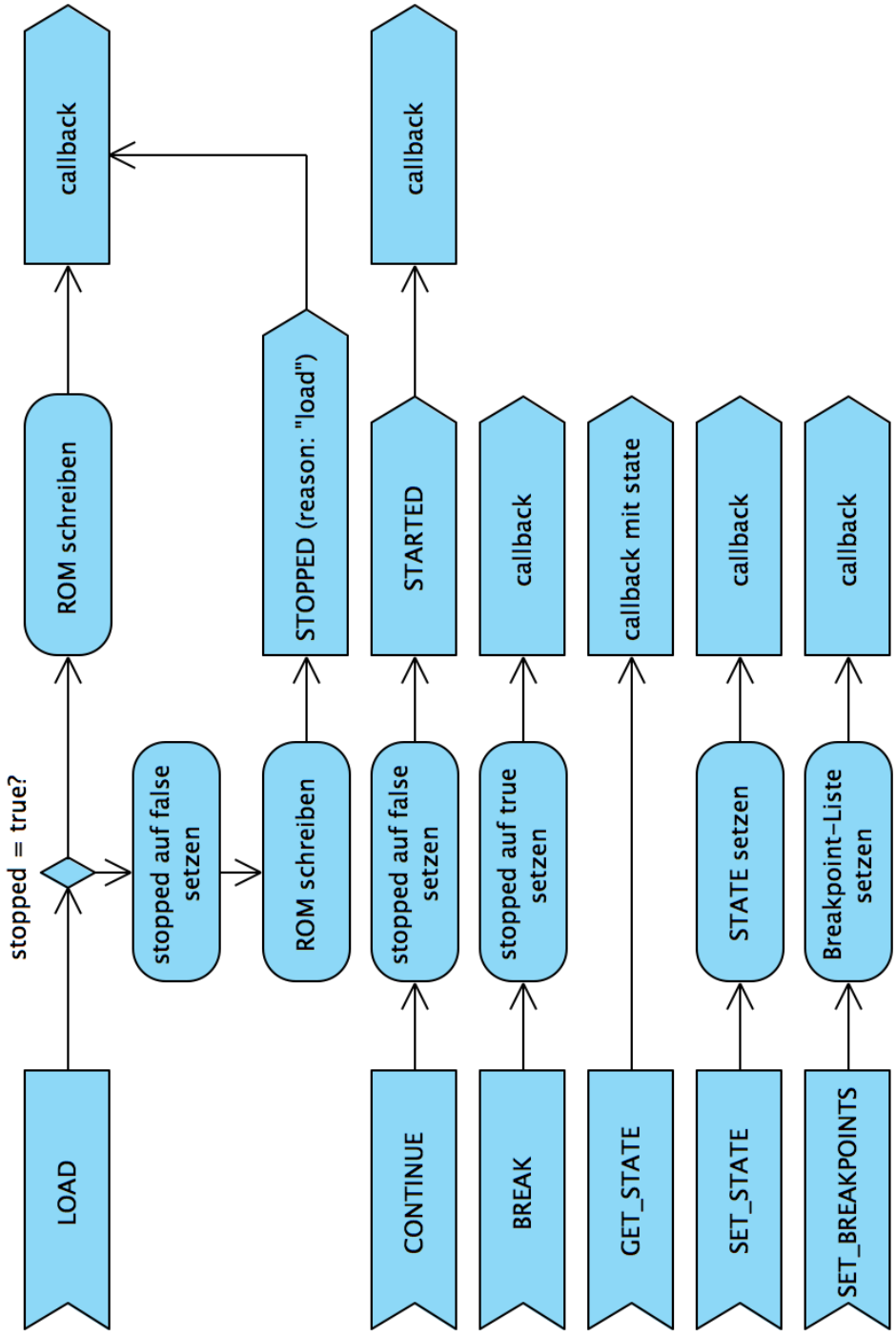
- Potentiometer
  - an Pin 10, type adc, mode in

Anhang 5: Emulatoraktivitätsdiagramme

Generelle Funktionsweise

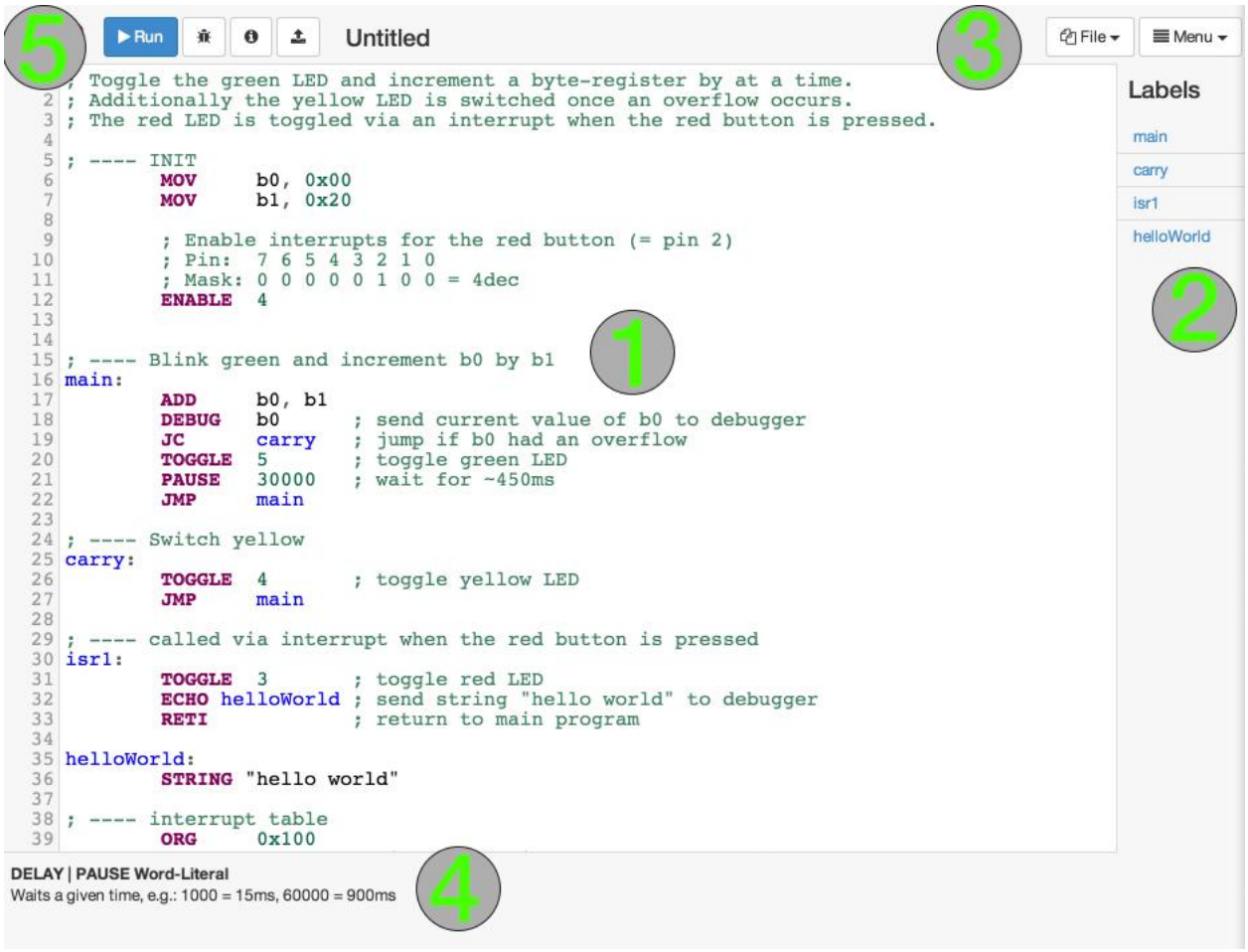


Action-Behandlung



Benutzerhandbuch

IDE

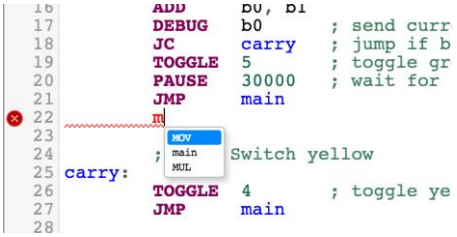



```
; Toggle the green LED and increment a byte-register by at a time.
; Additionally the yellow LED is switched once an overflow occurs.
; The red LED is toggled via an interrupt when the red button is pressed.

; ---- INIT
        MOV     b0, 0x00
        MOV     b1, 0x20

        ; Enable interrupts for the red button (= pin 2)
        ; Pin:  7 6 5 4 3 2 1 0
        ; Mask: 0 0 0 0 0 1 0 0 = 4dec
        ENABLE  4


; ---- Blink green and increment b0 by b1
main:
        ADD     b0, b1
        DEBUG   b0      ; send current value of b0 to debugger
        JC      carry   ; jump if b0 had an overflow
        TOGGLE  5       ; toggle green LED
        PAUSE   30000   ; wait for ~450ms
        JMP     main

; ---- Switch yellow
carry:
        TOGGLE  4       ; toggle yellow LED
        JMP     main

; ---- called via interrupt when the red button is pressed
isr1:
        TOGGLE  3       ; toggle red LED
        ECHO helloWorld ; send string "hello world" to debugger
        RETI            ; return to main program

helloWorld:
        STRING "hello world"

; ---- interrupt table
        ORG     0x100
```

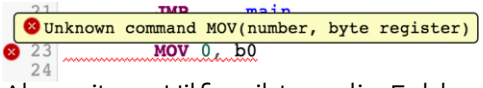
1: Texteditor

Dies ist das Herzstück der IDE. Es ist die Komponente mit der Quelltext eingegeben wird. Während man Code mit der Tastatur eintippt wird dieser in Echtzeit eingefärbt (Syntaxhighlighting) wodurch es einfacher ist Befehle, Parameter, Labels und Kommentare voneinander zu unterscheiden.

Zusätzlich gibt es eine Einrückungsfunktion, welche den Quelltext entsprechend einrückt: Befehle sind auf Ebene 1. Der `END` Befehl auf Ebene 0 und Kommentare auf beliebiger Einrückungsstufe.

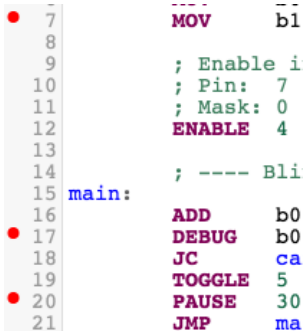


Während man Quelltext in das Fenster eingibt, hilft die kontextsensitive Autovervollständigung dem Programmierer oder der Programmiererin die richtigen Befehle und Parameter zu finden. Hierzu werden in einem Fenster unter dem aktuellen Cursor die möglichen Texteingaben angezeigt.



Als weitere Hilfe gibt es die Fehleranzeige, welche dem Benutzer oder der Benutzerin bereits während dem Tippen, Fehler im Code anzeigt und diese genau beschreibt - und das, ohne vorher kompilieren zu müssen.

Weiters unterstützt die IDE das Setzen von Haltepunkten (Breakpoints). Dieses sind spezielle Markierungen im Quelltext die dem Debugger die Information geben, wann angehalten werden soll. Dies ist vor allem dann praktisch, wenn man nach Fehlern sucht oder Programme analysieren muss.
Das Setzen von Haltepunkten ist einfach: Durch Klick auf die Zeilennummer wird ein roter Punkt angezeigt, der den Haltepunkt darstellt. Ein weiterer Klick entfernt diesen Haltepunkt wieder.



2: Label Liste

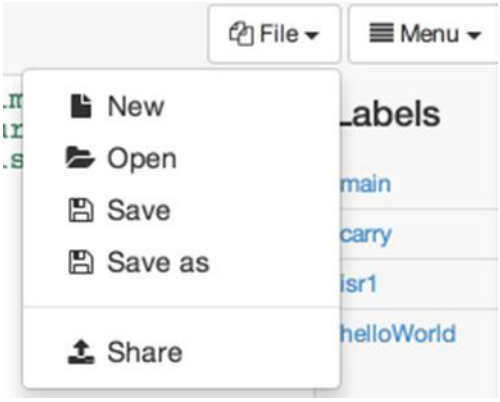
Diese Liste zeigt alle Labels die im aktuellen Quelltext definiert wurden an. Dies ist vor allem für längere Sourcecodes praktisch, da hierdurch die Übersicht bewahrt werden kann. Weiters ist es möglich durch Klick auf ein bestimmtes Label den Cursor direkt zur Definition springen lassen zu können - wenn nötig wird auch an die entsprechende Stelle im Codeeditor gescrollt.

3: Menü

Das Menü von *yasp* ist in zwei Teile aufgeteilt: File und Menu. Dadurch bleibt es übersichtlich und jeder Nutzer findet schnell die notwendigen Aktionen.

File

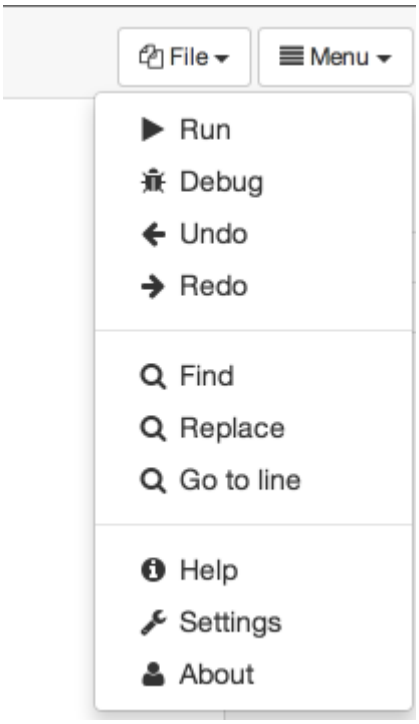
Im File Menü sind die FileManager Operationen zugänglich. Dazu gehört: Speichern, Speichern unter, Öffnen, Neu und Teilen.



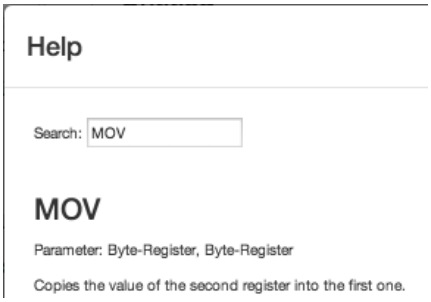
- Save: Öffnet den Filedialog im "Speichern" Modus. Das heißt: Wenn die Datei bereits einen Namen erhalten hat, wird die Datei direkt gespeichert, ansonsten wird der User aufgefordert einen Dateinamen festzulegen.
- Save As: Hier wird in jedem Fall der User aufgefordert einen Dateinamen festzulegen, auch wenn diese bereits einen Namen hat. Öffnet ebenfalls den Dateidialog im "Speichern" Modus.
- New: Erstellt eine neue Datei. Der User wird hierbei aufgefordert den Dateinamen festzulegen. Öffnet den Dateidialog im "Neu" Modus.
- Open: Öffnet eine existierende Datei. Der Dateidialog erscheint im "Open" Modus.
- Share: Teilt eine Datei. Hierbei wird der Quelltext auf einen Server hochgeladen und der User erhält einen Link, den er mit anderen teilen kann. Somit erhalten sie ebenfalls Zugriff auf den Quelltext.

Menu

Im Menüpunkt "Menu" lassen sich alle anderen Aktionen durchführen:



- **Run:** Führt das aktuelle Programm aus und öffnet den Debugger.
- **Debug:** Führt das aktuelle Programm aus, stoppt aber an der ersten Instruktion im Debugger.
- **Undo:** Rückgängig Funktion. Die letzte Aktion wird ungeschehen gemacht.
- **Redo:** Wiederholen Funktion. Die letzte Aktion wird wiederhergestellt.
- **Find:** Findet einen Text im Quelltext. Diesen muss man in das gegebene Textfeld eingeben und durch Eingabe bestätigen.
- **Replace:** Ersetzt einen Text im Quelltext durch einen anderen. Hierbei wird im ersten Textfeld der Suchstring und im zweiten Textfeld der Ersetzungsstring eingegeben.
- **Go to line:** Springt zur angegebenen Zeile im Texteditor. Scrollt, wenn notwendig, zur gesuchten Zeile.
- **Help:** Öffnet die Dokumentation über alle Befehle in einem Dialogfenster. In diesem Fenster kann zusätzlich nach einem bestimmten Befehl über die Suchfunktion gesucht werden. Siehe Bild.
- **Settings:** Öffnet das Einstellungsmenü mit dem das Verhalten der IDE modifiziert werden kann.
- **About:** Öffnet den "Über uns" Dialog, der Informationen über *yasp* und das Entwicklerteam bereitstellt.



4. Schnellhilfe

Die Schnellhilfe ist eine Dokumentation, die vor allem für Anfänger eine große Hilfe darstellt. Während man den Befehl eintippt liefert sie nützliche Informationen, wie zum Beispiel Parameter und Beschreibung, die dem Programmierer oder der Programmiererin viel Arbeit abnehmen. Wenn nicht eindeutig ist, welcher Befehl gesucht ist, liefert die Schnellhilfe eine Liste möglicher Befehle.

5. Aktionsmenü

Das Aktionsmenü besteht aus vier Buttons, die alle bereits im regulären Menü erreichbar sind, jedoch so häufig gebraucht werden, dass es praktischer ist, wenn diese direkt sichtbar sind.

Führt das aktuelle Programm aus und öffnet den Debugger.

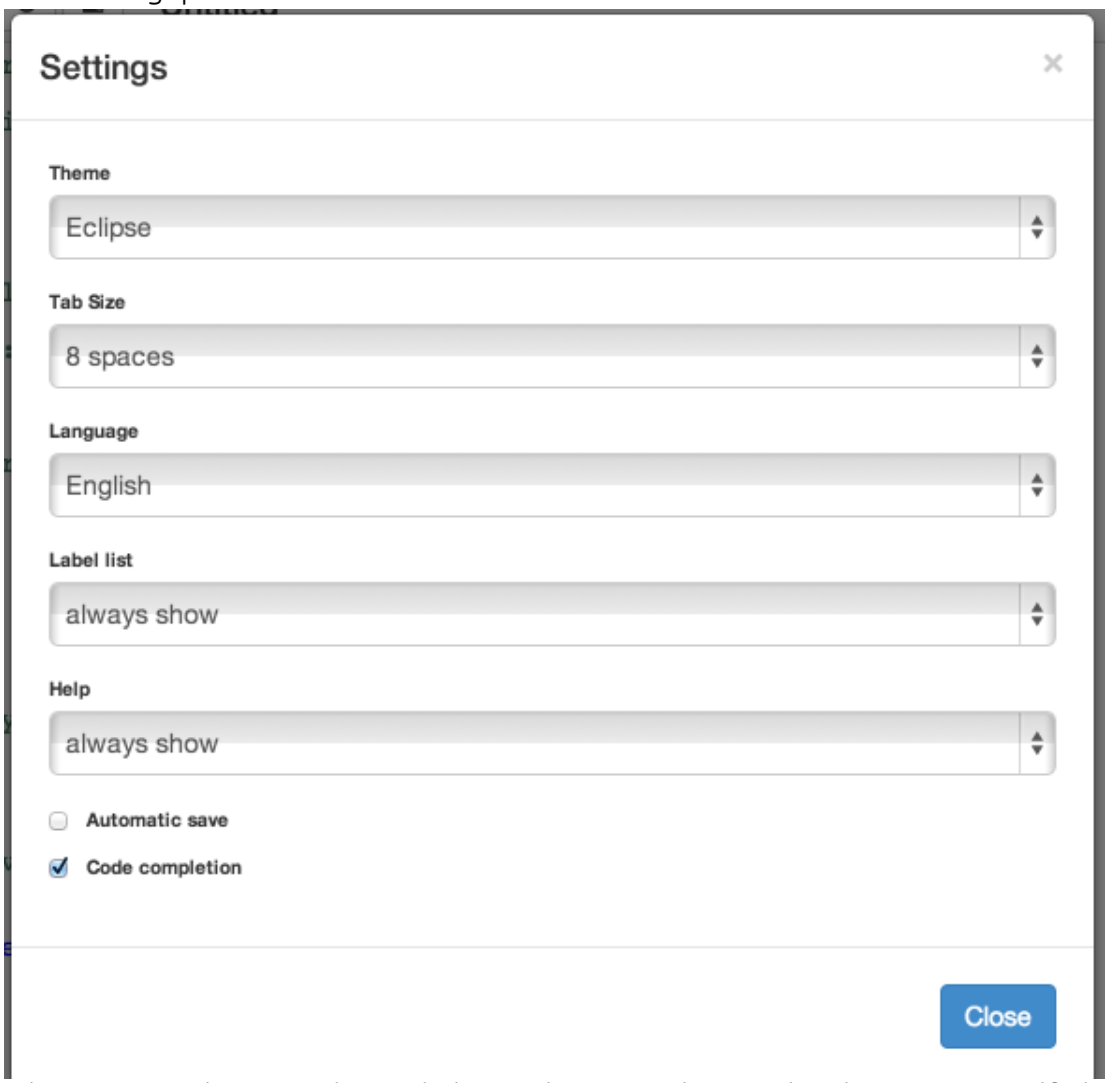
Führt das aktuelle Programm aus, stoppt aber an der ersten Instruktion im Debugger.

Öffnet die Dokumentation über alle Befehle in einem Dialogfenster.

Teilt eine Datei.

Einstellungen

Im Einstellungsdialog wird das Verhalten der IDE kontrolliert und es kann nach eigenen Vorlieben und Interessen angepasst werden.



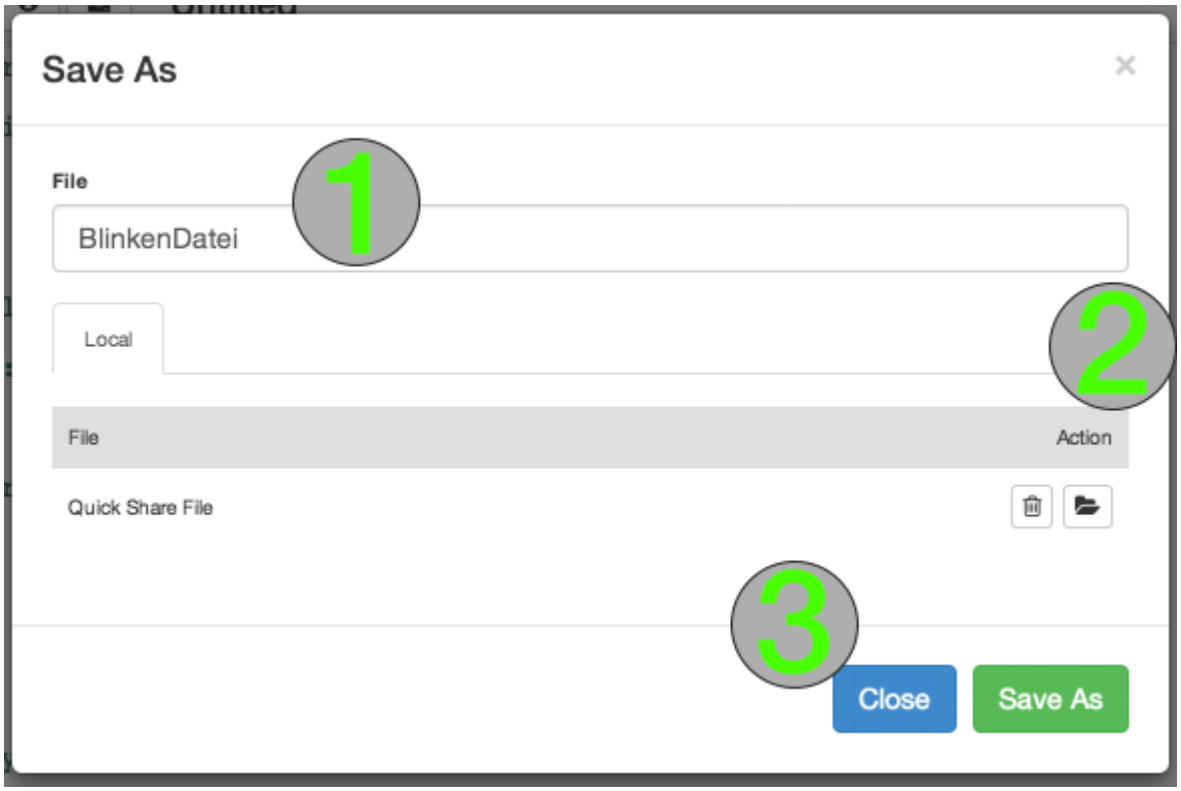
- **Theme:** Setzt das Syntaxhervorhebungsthema. Hierbei werden die Hintergrundfarben, Kommentarfarben, Befehlsfarben und mehr verändert.
- **Tab Size:** Setzt die Tab Größe im Texteditor.
- **Language:** Setzt die Sprache in *yasp*. Momentan sind die Sprachen Deutsch und Englisch implementiert. Andere Sprachen sind jedoch möglich.
- **Label list:** Setzt die Darstellungsart der Labelliste. Hierbei zeigt "always show" die Labelliste immer an, "show if needed" nur wenn sie gebraucht wird (Mauszeiger ist über der Liste) und "never show" zeigt die Liste nie an.
- **Help:** Setzt die Darstellungsart der Schnellhilfe. Hierbei zeigt "always show" die Schnellhilfe immer an, "show if needed" nur wenn sie gebraucht wird (Mauszeiger ist über der Hilfe) und "never show" zeigt die Hilfe nie an.
- **Automatic Save:** Aktiviert oder deaktiviert das automatische Speichern des Quellcodes. Wenn das automatische Speichern aktiviert ist, wird der Quelltext regelmäßig lokal in der Datei "Automatic Save" zwischengespeichert.
- **Code Completion:** Aktiviert oder deaktiviert die Autovervollständigung. Sie kann jedoch selbst im deaktivierten Zustand durch Steuerung+Leertaste aufgerufen werden.

FileManager

Der FileManager stellt alle gängigen Dateioperationen zur Verfügung und dient zur einfachen Verwaltung der Daten. Die Daten werden im `localStorage` des Browsers gespeichert, sofern man sich im Local Tab befindet und auf einem Server, falls die Selektion am Server tab ist.

Speichern Unter

Der „Speichern Unter" Dialog wird in jedem Fall geöffnet, auch wenn die Datei bereits einen Namen hat.



1: Name Eingabefeld

In diesem Textfeld ist der Dateiname einzugeben, der dann übernommen wird.

2: Dateiliste

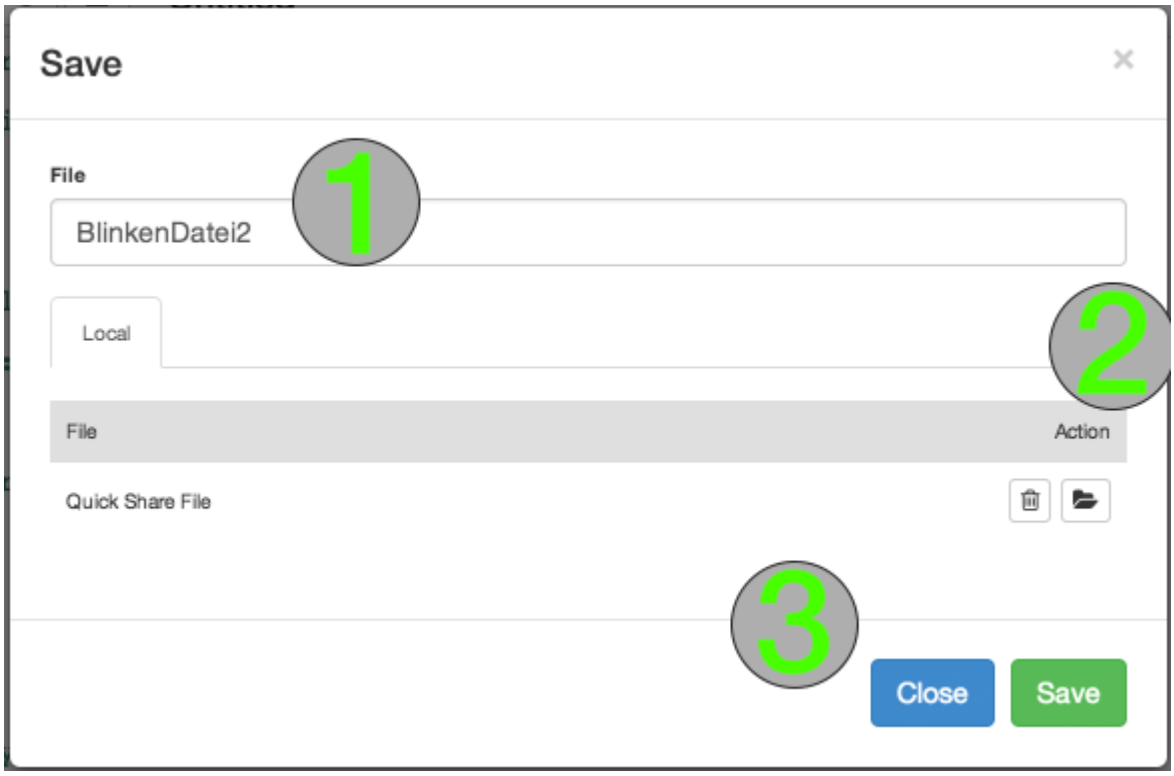
In dieser Liste sind alle gespeicherten Dateien sichtbar. Zusätzlich lässt sich durch das Löschen Symbol ( ) eine Datei aus der Liste entfernen. Das Öffnen Symbol ( ) kopiert den Namen der Datei in das Eingabefeld.

3: Button

Der Close Button bricht die aktuelle Aktion ab. Der Save As Button speichert die Datei unter dem angegebenen Namen. Danach wird sie geöffnet.

Speichern

Der „Speichern" Dialog wird nur dann geöffnet, falls die Datei gespeichert wird, jedoch keinen Namen hat.



1: Name Eingabefeld

In diesem Textfeld ist der Dateiname einzugeben, der dann übernommen wird.

2: Dateiliste

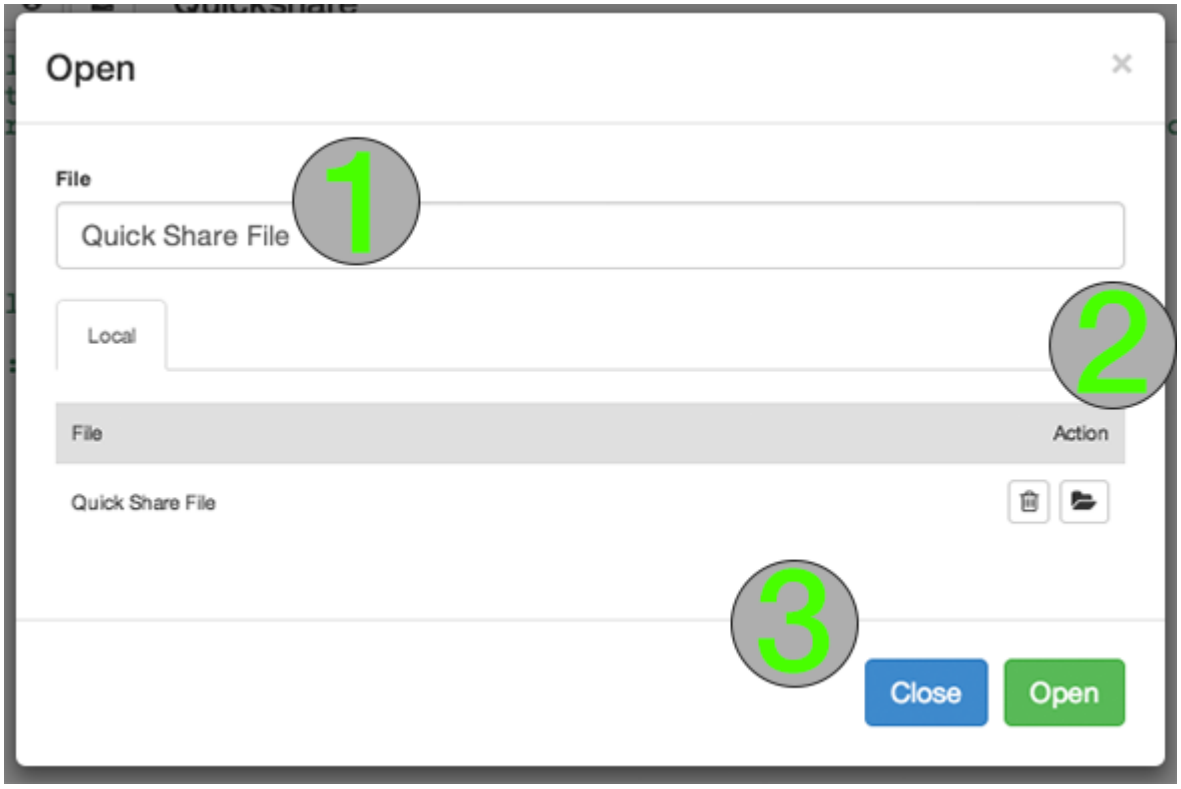
In dieser Liste sind alle gespeicherten Dateien sichtbar. Zusätzlich lässt sich durch das Löschen Symbol ( ) eine Datei aus der Liste entfernen. Das Öffnen Symbol ( ) kopiert den Namen der Datei in das Eingabefeld.

3: Button

Der **Close** Button bricht die aktuelle Aktion ab. Der **Save** Button speichert die Datei unter dem angegebenen Namen. Danach wird sie geöffnet.

Öffnen

Der „Öffnen" Dialog öffnet eine Datei und schließt, falls vorhanden, die alte Datei.



1: Name Eingabefeld

In diesem Textfeld ist der Dateiname einzugeben, der dann übernommen wird.

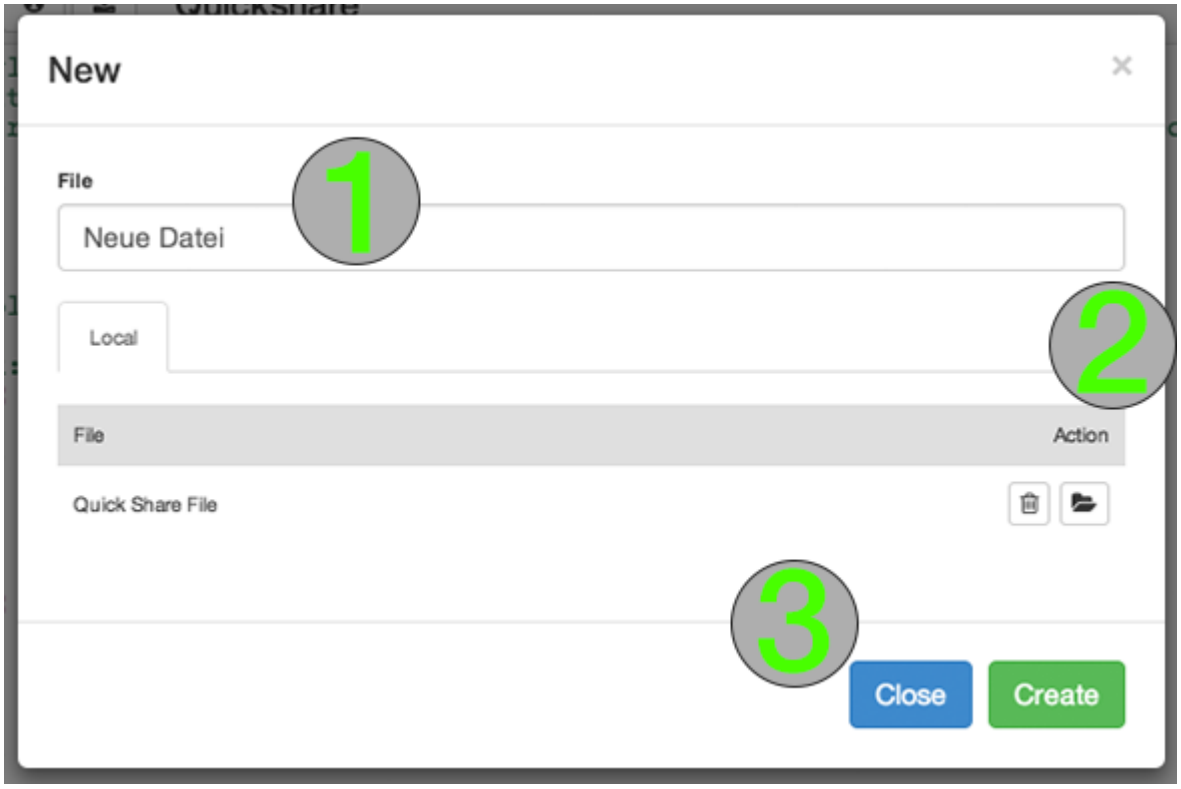
2: Dateiliste

In dieser Liste sind alle gespeicherten Dateien sichtbar. Zusätzlich lässt sich durch das Löschen Symbol ( ) eine Datei aus der Liste entfernen. Das Öffnen Symbol ( ) kopiert den Namen der Datei in das Eingabefeld.

3: Button

Der **Close** Button bricht die aktuelle Aktion ab. Der **Open** Button öffnet die entsprechend ausgewählte Datei in der IDE.

Neu
Der „Neu“ Dialog erstellt eine Datei und speichert sie.



1: Name Eingabefeld
In diesem Textfeld ist der Dateiname einzugeben, der dann übernommen wird.

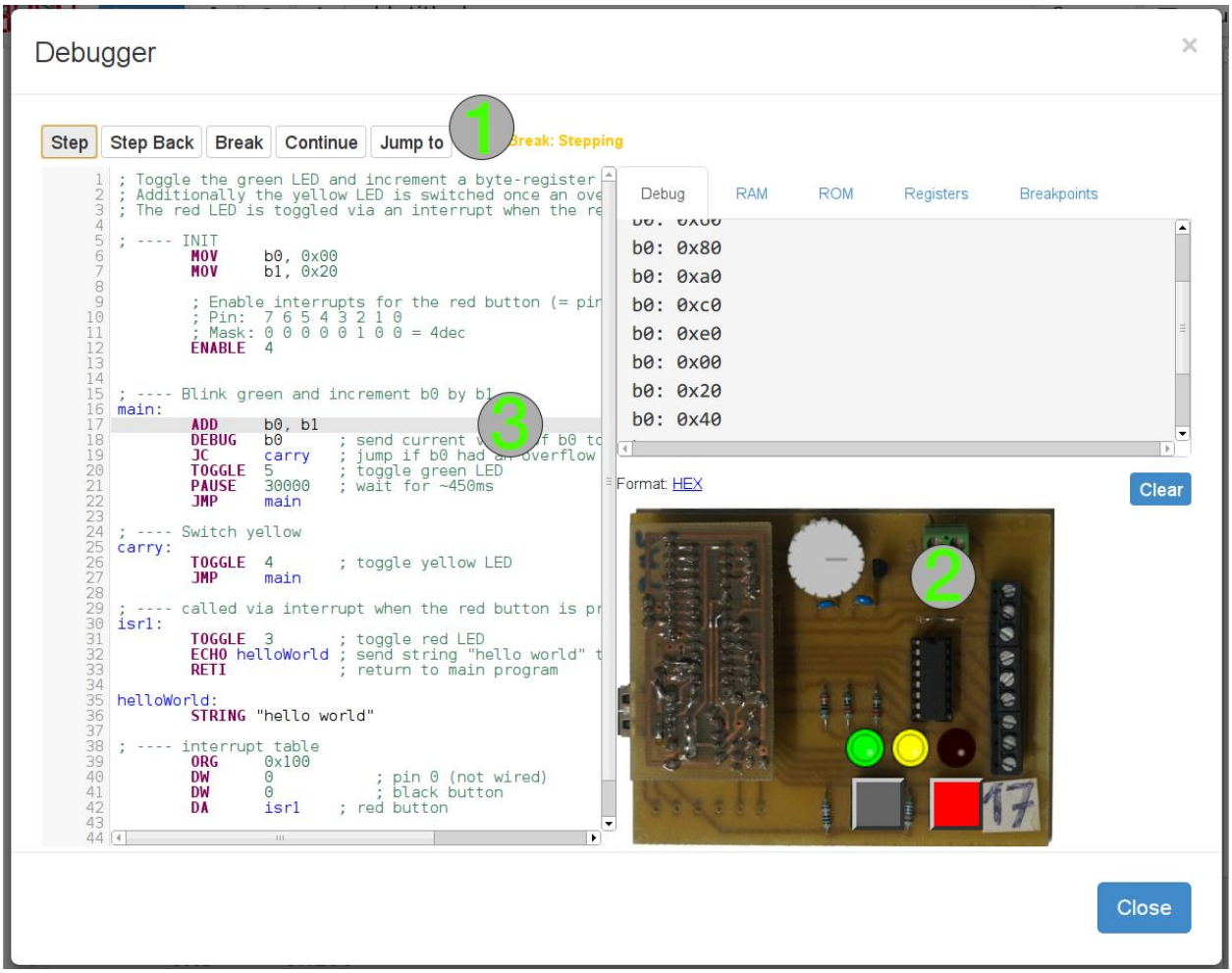
2: Dateiliste
In dieser Liste sind alle gespeicherten Dateien sichtbar. Zusätzlich lässt sich durch das Löschen Symbol ( ) eine Datei aus der Liste entfernen. Das Öffnen Symbol ( ) kopiert den Namen der Datei in das Eingabefeld.

3: Button
Der **Close** Button bricht die aktuelle Aktion ab. Der **Create** Button erstellt die entsprechende Datei und öffnet sie in der IDE.

Server Login

Wenn man Dateien am Server speichern will, so muss man sich vorher am Server einloggen. Hierbei verwendet man seinen LDAP Useraccount. Die Dateien werden danach beim Speichern auf den Schulservern geladen.

Debugger-Hauptfenster



1: Buttons und Status

- Step
  Führt die nächste Instruktion aus und hält die Ausführung anschließend wieder an.
- Step Back
  Springt zu der zuletzt ausgeführten Instruktion. Dies ist nur möglich wenn vor **Step Back** ein **Step** ausgeführt wurde.
- Break
  Hält die Ausführung an und wartet auf weitere Befehle, wie zum Beispiel ein **Step**.
- Continue
  Führt die Ausführung ohne Einschränkungen aus. Es werden so lange Instruktionen ausgeführt bist der Emulator durch den **Break**-Befehl angehalten wird.
- Jump To
  Setzt die aktuelle Position der Ausführung. Hierzu wird aus dem Mauscursor ein Fadenkreuz mit dem die gewünschte Zeile ausgewählt werden kann.
- Status
  Zeigt den aktuellen Status des Emulators an. Folgende Möglichkeiten gibt es:
  - Running
    Der Emulator läuft in Echtzeit.

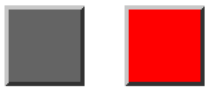
- Break
  - Stepping
    Der Emulator führt eine Instruktion nach der anderen aus, siehe **Step**.
  - BREAK-Instruction
    Der Emulator wurde durch eine `BREAK`-Instruktion angehalten.
  - Divided by zero
    Es wurde versucht mit dem `DIV`-Befehl durch Null zu teilen.
  - Debugger
    Der Emulator wurde durch das **Break**-Kommando angehalten.
  - Invalid instruction
    Es wurde versucht eine ungültige Instruktion auszuführen. Dies kann entweder auf einen Fehler in yasp selbst hinweisen oder es wurden mit der `WRROM`-Instruktion ungültige Daten in den ROM geschrieben.
  - Breakpoint
    Ein Breakpoint wurde erreicht. Die Ausführung kann mit **Step** oder **Continue** fortgeführt werden.

2: Hardware

Bei 2. wird der aktuelle Status der Hardware angezeigt. In der Standardkonfiguration stehen folgende Elemente zur Verfügung:

Potentiometer
Ein Drehregler welcher im Emulator mit den `ADC`-Instruktionen ausgelesen werden kann. Er meldet einen analogen Wert von 0 bis 255 an den Emulator zurück. Standardmäßig ist dieser an Pin 11 angeschlossen.

LEDs
Leuchtdioden in 3 Farben welche im Emulator mit den `HIGH`, `LOW` und `TOGGLE`-Instruktionen angesprochen werden können. Es ist eine rote (Pin 3), eine gelbe (Pin 4) und eine grüne (Pin 5) LED verfügbar.



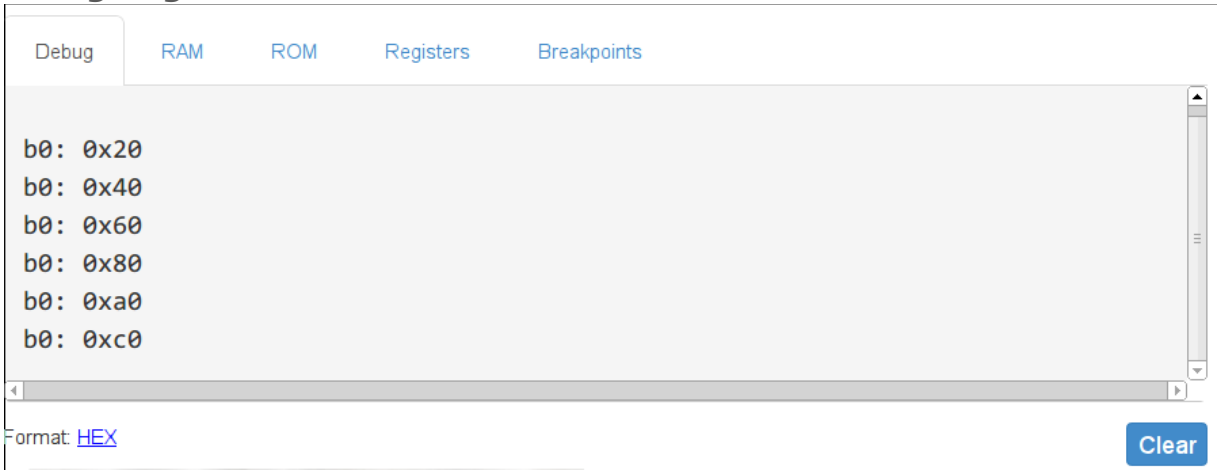
Buttons
Ein grauer und ein roter Button an Pin 1 und 2. Diese können mit der `PIN`-Instruktion ausgelesen werden. Zusätzlich ist es möglich Interrupts zu triggern, was im "Tutorial" der Technischen Dokumentation näher beschrieben wird.

3: Sourcecode

Anzeige des aktuell ausgeführten Sourcecodes. Durch einen Klick auf eine Zeilennummer kann ein Breakpoint gesetzt oder entfernt werden. Falls sich der Emulator im Stepping-Modus befindet wird das nächste Statement grau hinterlegt.

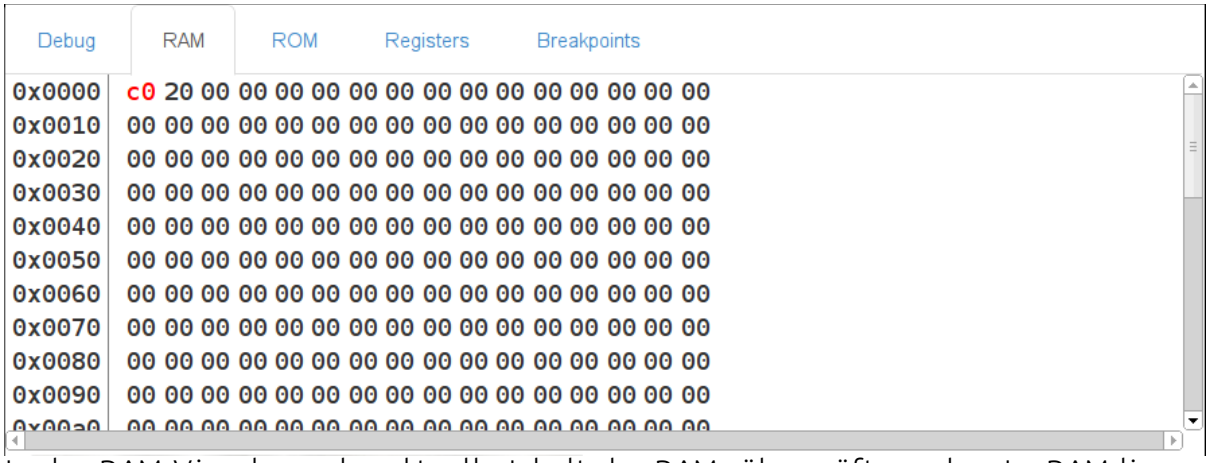
Debugger-Tabs

Debug-Log



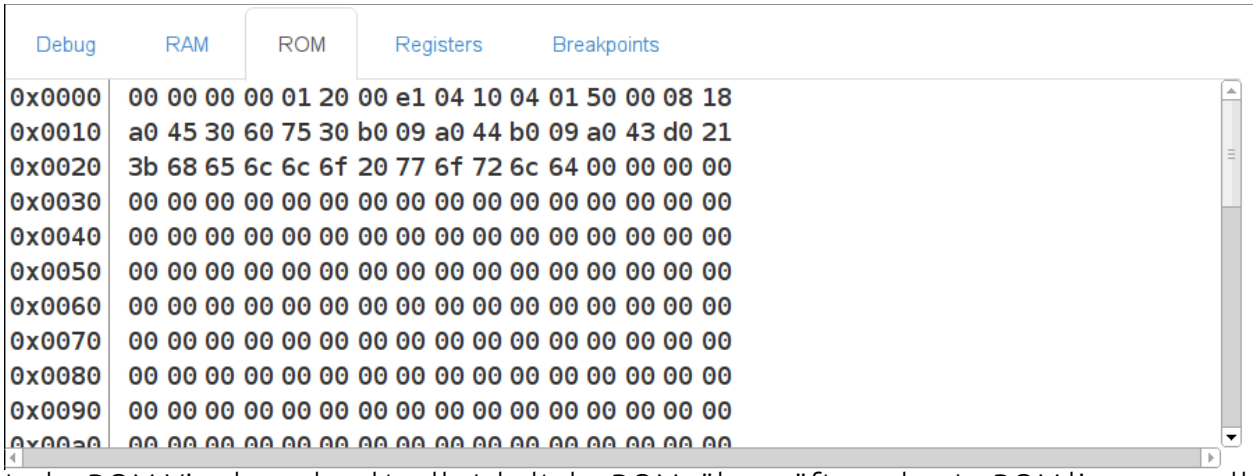
Im Debug-Log werden die Ausgaben der `DEBUG` und `PRINT`-Instruktionen angezeigt. Standardmäßig werden alle Register-Werte in im Hexadezimal-Format ausgegeben, dies kann mit dem **Format**-Switch geändert werden. Mit dem **Clear**-Button kann das Log geleert werden. Dieser Tab wird auch aktualisiert wenn sich das Programm im run-Modus befindet, also nicht gesteppt wird.

RAM-View



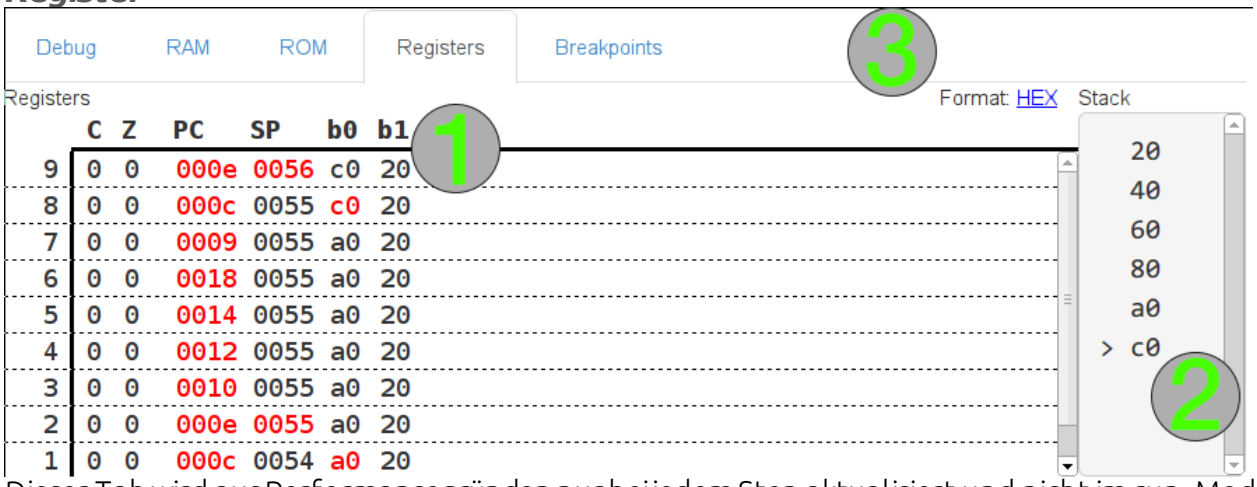
In der RAM-View kann der aktuelle Inhalt des RAMs überprüft werden. Im RAM liegen unter anderem die Werte der Register. Änderungen seit dem letzten Step werden rot markiert. Dieser Tab wird aus Performancegründen nur bei jedem Step aktualisiert und nicht im run-Modus.

ROM-View



In der ROM-View kann der aktuelle Inhalt des ROMs überprüft werden. Im ROM liegen vor allem der assemblierte Programmcode welcher in der Abbildung sichtbar ist. Zusätzlich befindet sich ab `0x100` die Interrupttabelle welche ebenfalls überprüft werden kann. Änderungen seit dem letzten Step werden rot markiert. Dieser Tab wird aus Performancegründen nur bei jedem Step aktualisiert und nicht im run-Modus.

Register



Dieser Tab wird aus Performancegründen nur bei jedem Step aktualisiert und nicht im run-Modus.

1: Registerwerte

Hier sind die Werte der Flags, der SPRs (Special Purpose Registers) und aller verwendeten GPRs (General Purpose Registers) zu sehen. Bei jedem Step wird eine Zeile in der History hinzugefügt, so dass sich Änderungen mit Hilfe der Einfärbung leicht nachvollziehen lassen.

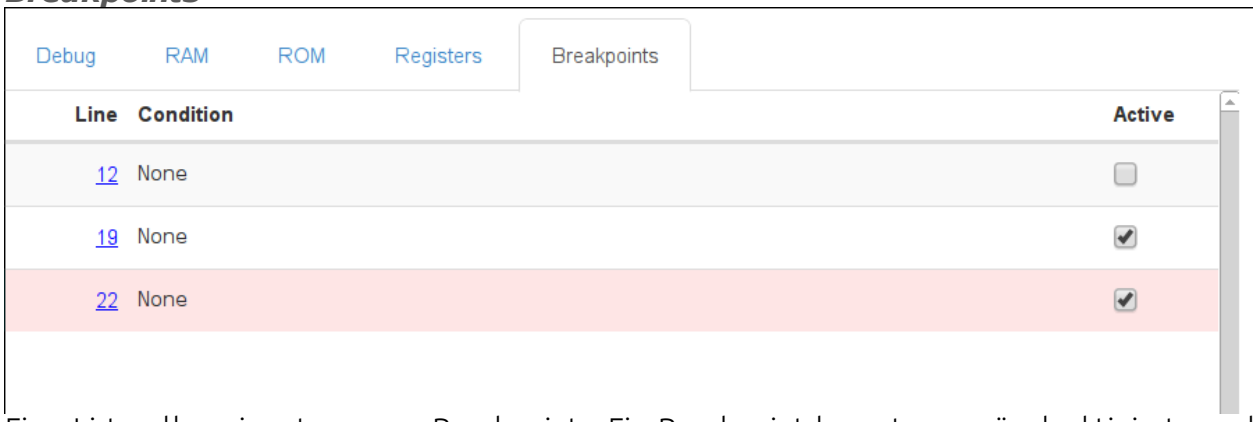
2: Stack

Der Stack, welcher hier von oben nach unten wächst, ist ebenfalls sichtbar. Das oberste Element wird durch einen Pfeil markiert.

3: Format

Mit dem Format-Switch lässt sich das Ausgabeformat der Registerwerte und des Stacks anpassen, so dass beispielsweise Bitshift-Operationen leichter nachvollziehbar sind.

Breakpoints

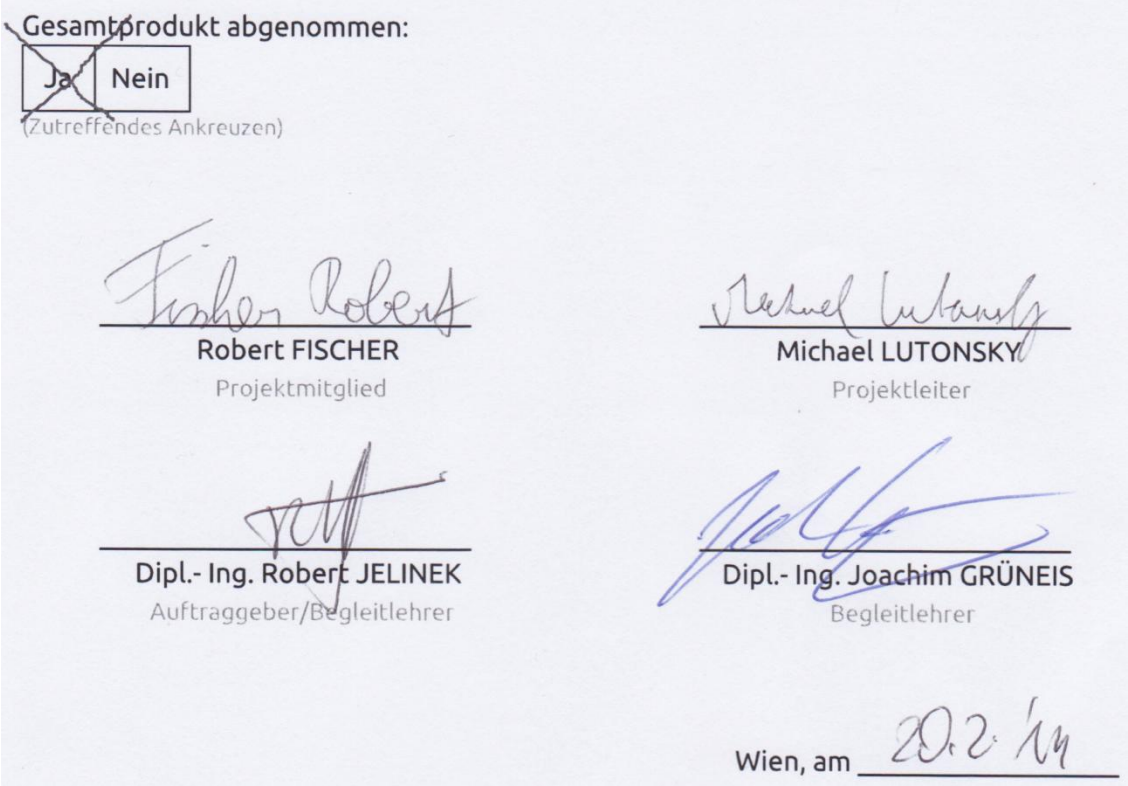


Eine Liste aller eingetragenen Breakpoints. Ein Breakpoint kann temporär deaktiviert werden indem man seine Checkbox betätigt, er wird dann in grau dargestellt. Falls das Programm gerade durch einen Breakpoint angehalten wurde, wird dieser rot hinterlegt. Ein Klick auf die Zeilennummer zeigt die Zeile im Editor an.

Abnahme

Kategorie	Testfall	KO	OK?	Kommentar
Emulator / Assembler	Testprogramm 01	Ja	✓	
Emulator / Assembler	Testprogramm 02	Ja	✓	
Emulator / Assembler	Testprogramm 03	Ja	✓	
Emulator / Assembler	Testprogramm 04	Ja	✓	
Emulator / Assembler	Testprogramm 05	Ja	✓	
Emulator / Assembler	Testprogramm 06	Ja	✓	
Emulator / Assembler	Testprogramm 07	Ja	✓	
Emulator / Assembler	Testprogramm 08	Ja	✓	
Emulator / Assembler	Testprogramm 09	Ja	✓	
Emulator / Assembler	Testprogramm 10	Ja	✓	
Emulator	Unit Tests	Ja	✓	
Assembler	Unit Tests	Ja	✓	
IDE	Öffnen	Ja	✓	
IDE	Schließen	Ja	✓	
IDE	Einstellungen verändern	Nein	✓	
IDE	Quelltext eingeben	Ja	✓	
IDE	Suchen	Nein	✓	suchauswahl entfernen!
IDE	Ersetzen	Nein	✓	

IDE	Gehe zu Zeile	Nein	✓	
IDE	Schnellhilfe	Nein	✓	
IDE	Hilfe	Nein	✓	volltext/befehlssuche
IDE	Speichern	Ja	✓	
IDE	Laden	Ja	✓	
IDE	Rückgängig	Ja	✓	
IDE	Wiederholen	Ja	✓	
Debugger	Break	Ja	✓	
Debugger	Step	Ja	✓	
Debugger	Step Back	Nein	✓	
Debugger	Continue	Ja	✓	
Debugger	PC setzen	Nein	✓	
~~Debugger~~	~~Execute~~	~~Nein~~		
Debugger	Breakpoint setzen	Ja	✓	
Debugger	Breakpoint entfernen	Ja	✓	
Debugger	Breakpoint temporär deaktivieren	Nein	✓	
~~Debugger~~	~~Bedingte~~ Breakpoints setzen	~~Nein~~		
Debugger	Zeilenbezogene	Nein		
	Bedingte Breakpoints			


Gesamtprodukt abgenommen:
Ja
Nein
(Zutreffendes Ankreuzen)
Robert FISCHER
Projektmitglied
Michael LUTONSKY
Projektleiter
Dipl.- Ing. Robert JELINEK
Auftraggeber/Begleitlehrer
Dipl.- Ing. Joachim GRÜNEIS
Begleitlehrer
Wien, am 20.2.14

Veröffentlichung

Website

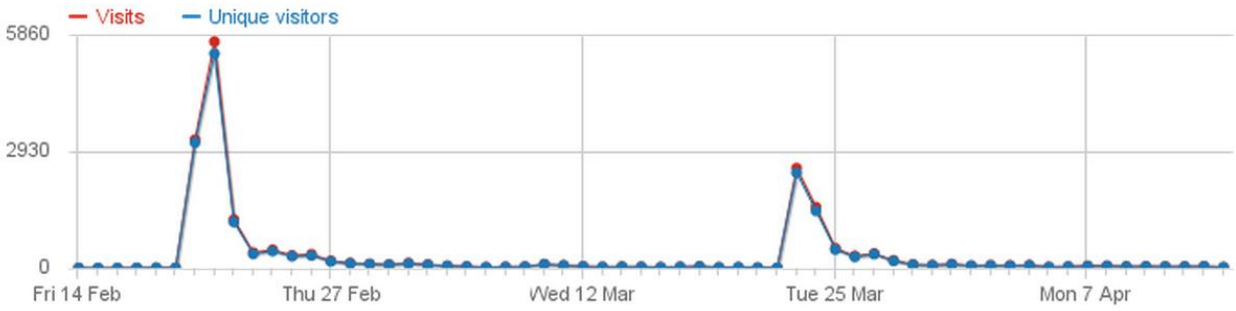
Zur Veröffentlichung von yasp haben wir eine Website mit den wichtigsten Fakten und Funktionen erstellt. Die Website wurde technisch sehr simpel gestaltet und kam völlig ohne dynamische Elemente aus, weswegen der Gesamtaufwand hierfür nur 2,5 Stunden betrug. Um eine möglichst große Zielgruppe erreichen zu können, wurde die gesamte Website auf Englisch verfasst. Insgesamt wurde die Seite im Zeitraum von Februar bis Anfang April 25.000 mal von Usern aus der ganzen Welt aufgerufen, wie im Punkt "Besucherstatistiken" zu sehen ist.

Link: http://yasp.me/
Demo: http://demo.yasp.me/

Besucherstatistiken

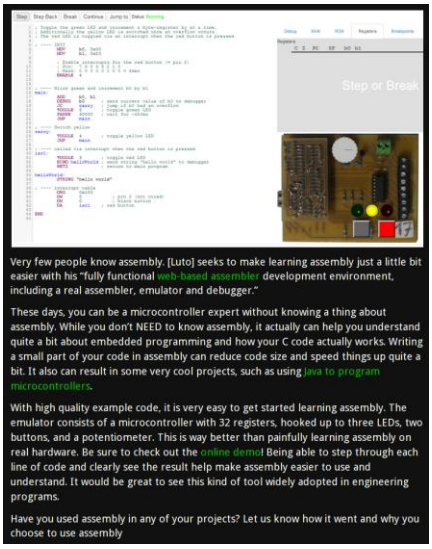
Besucheranzahl

Die Website des Projekts wurde insgesamt 25.000 mal aufgerufen. Bei der Auswertung der Aufrufe fallen zwei Spikes ins Auge, welche in der folgenden Grafik gut sichtbar sind. Die Aufrufe in den "flachen" Teilen betragen circa 30-40 Besucher pro Tag, wo sich die Zahlen auch Anfang Mai 2014 noch befanden. Der erste Spike wurde von der Veröffentlichung auf "hackernews" (https://news.ycombinator.com/) verursacht, der zweite durch den Artikel auf "hackaday" (http://hackaday.com). Insgesamt waren wir mit der Anzahl der Besucher überrascht und sehr zufrieden.



Besucherherkunft

Hackaday



Very few people know assembly. [Luto] seeks to make learning assembly just a little bit easier with his "fully functional web-based assembler development environment, including a real assembler, emulator and debugger."

These days, you can be a microcontroller expert without knowing a thing about assembly. While you don't NEED to know assembly, it actually can help you understand quite a bit about embedded programming and how your C code actually works. Writing a small part of your code in assembly can reduce code size and speed things up quite a bit. It also can result in some very cool projects, such as using Java to program microcontrollers.

With high quality example code, it is very easy to get started learning assembly. The emulator consists of a microcontroller with 32 registers, hooked up to three LEDs, two buttons, and a potentiometer. This is way better than painfully learning assembly on real hardware. Be sure to check out the online demo! Being able to step through each line of code and clearly see the result help make assembly easier to use and understand. It would be great to see this kind of tool widely adopted in engineering programs.

Have you used assembly in any of your projects? Let us know how it went and why you choose to use assembly

Hackaday ist ein Online-Magazin, welches jeden Tag Hacks und jede Woche ein sogenanntes "How-To" veröffentlicht. Ein Hack ist in diesem Fall die Änderung eines Produktes oder einer Software zum Zweck der Bequemlichkeit, Neuheit, Funktion oder Kreativität. Hack a Day wurde 2004 als Web-Magazin für "Engadget" gegründet.
Wir haben unser Projekt bei hackaday eingereicht, woraufhin einer der Redakteure einen Artikel über *yasp* verfasste, welcher am 23. März 2014 veröffentlicht wurde.

Der vollständige Artikel:
„Very few people know assembly. Luto seeks to make learning assembly just a little bit easier with his "fully functional web-based assembler development environment, including a real assembler, emulator and debugger."

These days, you can be a microcontroller expert without knowing a thing about assembly. While you don't need to know assembly, it actually can help you understand quite a bit about embedded programming and how your C code actually works. Writing a small part of your code in assembly can reduce code size and speed things up quite a bit. It also can result in some very cool projects, such as using Java to program microcontrollers.

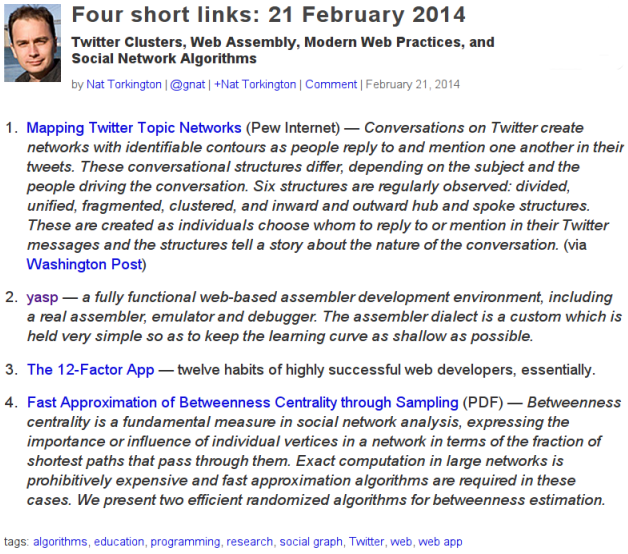
With high quality example code, it is very easy to get started learning assembly. The emulator consists of a microcontroller with 32 registers, hooked up to three LEDs, two buttons, and a potentiometer. This is way better than painfully learning assembly on real hardware. Be sure to check out the online demo! Being able to step through each line of code and clearly see the result help make assembly easier to use and understand. It would be great to see this kind of tool widely adopted in engineering programs."

Link: http://hackaday.com/2014/03/23/learning-assembly-with-a-web-based-assembler/

O'Reilly Radar

O'Reilly betreibt ein Fachblog in dem jeden Tag 4 neuartige Projekte in kurzer Form veröffentlicht werden. Am 21. Februar 2014 war *yasp* dabei. Dies war zusätzlich eine der ersten Reaktionen auf das Projekt in der Öffentlichkeit.

Link: http://radar.oreilly.com/2014/02/four-short-links-21-february-2014.html



Four short links: 21 February 2014

Twitter Clusters, Web Assembly, Modern Web Practices, and Social Network Algorithms

by Nat Torkington | @gnat | +Nat Torkington | Comment | February 21, 2014

1. Mapping Twitter Topic Networks (Pew Internet) — *Conversations on Twitter create networks with identifiable contours as people reply to and mention one another in their tweets. These conversational structures differ, depending on the subject and the people driving the conversation. Six structures are regularly observed: divided, unified, fragmented, clustered, and inward and outward hub and spoke structures. These are created as individuals choose whom to reply to or mention in their Twitter messages and the structures tell a story about the nature of the conversation.* (via Washington Post)
2. yasp — *a fully functional web-based assembler development environment, including a real assembler, emulator and debugger. The assembler dialect is a custom which is held very simple so as to keep the learning curve as shallow as possible.*
3. The 12-Factor App — twelve habits of highly successful web developers, essentially.
4. Fast Approximation of Betweenness Centrality through Sampling (PDF) — *Betweenness centrality is a fundamental measure in social network analysis, expressing the importance or influence of individual vertices in a network in terms of the fraction of shortest paths that pass through them. Exact computation in large networks is prohibitively expensive and fast approximation algorithms are required in these cases. We present two efficient randomized algorithms for betweenness estimation.*

tags: algorithms, education, programming, research, social graph, Twitter, web, web app

CoderRadio

Coder Radio ist ein Podcast welcher sich mit IT-Projekten und Softwareentwicklung beschäftigt. Das Projekt wurde in zwei Folgen behandelt:

- in Folge 92 (circa Minute 14), Interview über das Projekt mit dem Projektleiter: http://www.jupiterbroadcasting.com/53142/persona-non-grata-cr-92/ bzw. http://youtu.be/K4gyEmzWRc4?t=14m
- in Folge 90 (circa Minute 9), Erwähnung des Projektes und kurzes Review http://www.jupiterbroadcasting.com/52237/get-yourself-tested-cr-90/ bzw. http://youtu.be/lZzWvl2VQqo?t=9m45s

Blogs und Foren

Diverse Veröffentlichungen in Foren, sozialen Netzwerken und Blogs:

- Tweets https://twitter.com/search?q=yasp.me%20OR%20demo.yasp.me&src=typd&f=realtime
- Spanische Retro-Community http://www.zonadepruebas.com/viewtopic.php?f=15&t=4916
- Protogisisches Forum http://www.zonadepruebas.com/viewtopic.php?f=15&t=4916
- "Tuts 4 You"-Forum https://forum.tuts4you.com/topic/34778-yasp-web-based-assembleremulator-to-learn-assembly/
- cyberhades-Blog (Spanisch) http://www.cyberhades.com/2014/02/22/aprende-ensamblador-de-una-manera-facil-y-segura/
- geektheplanet-Blog (Spanisch) http://geektheplanet.net/7875/yasp-aprendiendo-ensamblador-en-linea.xhtml
- dtrinf-Blog (Spanisch) http://dtrinf.wordpress.com/2014/03/02/aprender-ensambrador-facilmente/
- unocero-Blog (Spanisch) http://www.unocero.com/2014/03/23/aprenda-ensamblador-desde-una-pagina-web/
- embedds-Blog http://www.embedds.com/learning-assembly-the-easy-way/

- Serious Trolling-Blog
  http://blog.puercopop.com/post/78006432044/yasp
- Another Word For It-Blog
  http://tm.durusau.net/?p=50889
- Digitrode-Blog (Russisch)
  http://digitrode.ru/theory/programming/110-yasp-brauzernyy-assembler-s-virtualnym-mikrokontrollerom.html

AXA-Wettbewerb

Der AxA Wettbewerb wurde 2012 ins Leben gerufen, um junge, zukünftige TechnikerInnen die Tür zu öffnen und ihre Ideen, Projekte und Forschungsarbeiten in einem wertschätzenden und fachkundigen Rahmen vorzustellen. Der Wettbewerb selbst richtet sich vor allem an Maschinenbau- und Hardwareprojekte. Wir haben uns ebenfalls bei diesem Wettbewerb beworben, sind allerdings auf Grund der Zielgruppe nicht in die zweite Runde gekommen.

Link: http://www.xtest.at/news/axa_teilnehmer_innen.html

Abschlussbericht

Ziele

In der nachfolgenden Tabelle findet sich die Zusammenfassung unserer Ziele. Es wurden alle SOLL-Ziele erfüllt und die Hälfte der KANN-Ziele. Zusätzlich ist anzumerken, dass die größeren KANN-Ziele vor allem als Anregungen für Folgeprojekte gedacht waren und sind.

Name	Typ	Erfüllt
Assembler implementieren	SOLL	Ja
Emulator implementieren	SOLL	Ja
Debugger implementieren	SOLL	Ja
Integrierte Entwicklungsumgebung im Browser	SOLL	Ja
Clientseitiges Speichern von Quellcode	SOLL	Ja
Modularer Aufbau (Assembler bzw. Emulator austauschbar)	SOLL	Ja
Mehrbenutzersystem	KANN	Nein
C-Like-Compiler mit Echtzeit Assembler Darstellung	KANN	Nein
Unterstützung mobiler Endgeräte (z.B.: Android oder iOS)	KANN	Teilweise
Serverseitiges Speichern von Quellcode	KANN	Ja
Unterstützung der Sound-Befehle	KANN	Nein
Unterstützung von Autovervollständigung in der IDE	KANN	Ja

Retrospektive

Frontend-Framework für die IDE

Die IDE wurde abgesehen von jQuery und Bootstrap in purem JavaScript geschrieben, da eine Anforderung im Lastenheft war so wenig externe Libraries wie möglich zu verwenden. Im Nachhinein betrachtet wäre eine Library wie zum Beispiel AngularJS sehr praktisch bei der Implementierung gewesen, da diese viel Arbeit, vor allem beim Anzeigen von dynamischen Werten wie im Debugger, abnimmt.

Unit-Tests

Für den Assembler und Emulator gibt es eine große Anzahl an Unittests, die die korrekte Übersetzung beziehungsweise Ausführung des Codes garantieren. Diese wurden jedoch nur auf Wunsch der Entwickler ausgeführt, was dazu führte, dass Fehler in den exotischeren Bereichen (zum Beispiel der `RANDOM`-Befehl) nicht sofort aufgefallen sind. Für weitere Projekte empfehlen

wir es sehr einen Build- bzw. Continuous-Integration-Server aufzusetzen, welcher die Tests automatisch ausführt und bei Fehlern Alarm gibt.

Sprintlänge streng einhalten

Die meisten unserer Sprints waren, wie auch geplant, eine Woche lang. Jedoch wurde die Sprintlänge gegen Ende des Projektes manchmal auf zwei Wochen erhöht, da auch der Aufwand an Hausübungen und Tests zunahm. Das führte zu einer lascheren Kontrolle der erledigten Arbeiten, da diese immer nur am Ende eines Sprints durchgeführt wurde. Wir hätten die Länge der Sprints streng einhalten sollen, selbst wenn das bedeutet, dass in manchen Wochen nur ein bis drei Stunden Arbeit verrichtet werden.

Technische Hindernisse bei der Umsetzung

Emulator-Performance in JavaScript

Der Emulator in yasp musste eine bestimmte Taktrate erreichen, um wirklich gut nutzbar zu sein. Da das gesamte Projekt in JavaScript realisiert wurde war dies eine kleine Herausforderung. Auf Grund des Sprach-Designs von JavaScript (zum Beispiel dynamic Typing) ist es keine besonders schnelle Programmiersprache. Einzig der Wettbewerb zwischen den Browsern hat sehr viele Optimierungen an den Implementierungen von JavaScript selbst hervorgebracht, welche ein rechenintensives Programm wie einen Emulator überhaupt erst möglich machen. Dennoch gibt es eine Reihe von Tipps und Tricks, die man beachten sollte.

`if(variable) { }`

Ein if-Statement kann in JavaScript leicht verwendet werden um zu überprüfen, ob eine Variable einen Wert enthält. Dieser Code ist sehr schnell zu schreiben und findet sich zu Hauf in "normalem" JavaScript-Programmen. Das Problem hierbei ist, dass in JavaScript viele Werte gibt, welche `false` entsprechen, wie zum Beispiel die Zahl `0`, ein leerer String (`""`), ein leeres Array (`[]`) oder sogar die Zahl 0 als String (`"0"`). Die Engine muss nun jeden dieser Werte mit der gegeben Variable vergleichen, was zu Performanceproblemen führen kann wenn dieser Code oft ausgeführt wird, wie in einem Emulator. Wenn man weiß welcher Wert genau gesucht wird (zum Beispiel `null` oder `undefined`) sollte immer der Identity-Operator und der gesuchte Wert verwendet werden (siehe auch "Identity- vs. Equality-Operator").

call() und apply()

Um in JavaScript den Wert von `this` für einen Funktionsaufruf zu setzen, stehen zwei Möglichkeiten zur Auswahl: `call` und `apply`. Aus Programmierersicht ist der einzige Unterschied, dass `call` die Parameter der Funktion direkt übergeben (`func.call({}, 1, 2)`) bekommt, `apply` jedoch als Array (`func.apply({}, [1, 2])`). Da der Emulator eine variable Anzahl von Parametern für die einzelnen Instruktionen unterstützen soll wurde die `apply`-Funktion gewählt. Es stellte sich jedoch schnell heraus, dass diese viel langsamer als `call` ist. Im Source-Code von v8 (die JavaScript-Engine von Google Chrome) ist zu sehen, dass `call` den bereits übergebenen Stack, welcher auch die Parameter enthält "recycled", `apply` jedoch den Stack aus der Array-Referenz komplett neu aufbauen muss, was um einiges länger dauert. Letztendlich wurde im Emulator die `call`-Funktion zusammen mit "Loop-Unrolling" eingesetzt.

Caching

Operationen auf Bit-Ebene wie zum Beispiel Shiften sind bei JavaScript nicht sehr performant, was daran liegt, dass die Sprache nur einen Datentyp für Zahlen kennt, den double. Somit muss die Zahl für jede Bit-Operation in einen Integer umgewandelt, die Operation angewendet und danach wieder zurückgewandelt werden. Diese Operationen werden beim Einlesen der Befehle im Emulator sehr oft verwendet, was zu einer schlechten Performance geführt hat. Die Lösung war

ein Cache der schon eingelesenen Befehle auf welchen sehr viel schneller zugegriffen werden kann, so dass die Bit-Operationen pro Befehl nur einmal zum Einsatz kommen.

Loop-Unrolling

Der Emulator sollte eine variable Anzahl von Parametern für die Instruktionen unterstützen. Zu diesem Zweck wurden die notwendigen Werte in einer Schleife geladen und zu einem Array hinzugefügt. Dieses Vorgehen unterstützt zwar eine beliebige Anzahl an Parametern, ist allerdings nicht sehr performant. Die letztendlich implementierte Variante unterstützt nur null, einen oder zwei Parameter, womit die Performance drastisch gesteigert werden konnte. Falls in der Zukunft mehr als zwei Parameter notwendig sein sollten, lässt sich der Code leicht erweitern.

Zugriffe auf Objekte und Arrays

Zugriffe auf Arrays sind, wie erwartet, im JavaScript weit schneller als Zugriffe auf Objekt-Attribute, auch wenn das Objekt nur numerische Indexe hat. Dies war zum Beispiel beim Caching der Instruktionen im Emulator wichtig.

Typed-Arrays

Seit einigen JavaScript-Versionen ist es möglich sogenannte "Typed Arrays" einzusetzen. Diese werden intern als genau der gewünschte Datentyp abgelegt und bieten somit eine weit bessere Performance als klassische JavaScript-Arrays. Im Emulator wurde vor allem vom `Uint8Array` (ein Array von bytes ohne Vorzeichen) gebrauch gemacht, um den RAM und ROM abzuspeichern.

Identity- vs. Equality-Operator

In Javascript gibt es zwei Operatoren und die Gleichheit zweier Werte zu überprüfen, den Equality-Operator (==) und den Identity-Operator (===). Der Equality-Operator überprüft zusätzlich auch Werte die nicht ganz gleich sind, aber eine gleiche Bedeutung haben. Zum Beispiel sind `0` und `false` mit == gleich. Das kann wie auch bei `if(variable)` auf Kosten der Performance gehen.

Firefox

Firefox ist bei vielen Funktionen relativ eigenwillig, so kann man nicht ohne Weiteres eine große Anzahl an Webworker-Messages oder Log-Messages absetzen ohne das Einfrieren des gesamten Browsers zu riskieren. Zusätzlich hat Firefox Probleme mit großen DOM-Elementen, wie zum Beispiel das Debug-Log in unserem Fall, in dem sich schnell mehrere tausend Nachrichten ansammeln können. Die Lösung ist hier oft Rate-Limiting wie im Sprint-Protokoll 0.9 beschrieben.

Simulation der Pulsweitenmodulation

PWM (Pulsweitenmodulation) ist einer der Hauptinhalte des Lehrstoffs und war damit unentbehrlich. Prinzipiell ist es einen Technik mit der man LEDs dimmen oder pulsieren lassen kann. Ursprünglich waren für die Simulation von PWM zwei Stunden Arbeitsaufwand eingeplant, jedoch stellte sich schnell heraus, dass wir hier noch größere Probleme haben werden, welche sich über mehrere Monate zogen.
Der erste Ansatz zählte die Anzahl der An- und Aus-Statusänderungen, berechnet das Verhältnis und bestimmte so die Leuchtkraft der LED. Dieser Lösungsweg wurde in dem Hardwareelement selbst realisiert. Leider führte er, je nach genauer Einstellung der Parameter, entweder zu Flickern oder einer viel zu langsamen Reaktion.
Nach diversen Reparaturversuchen auf Seiten beider Projektmitglieder wurde der erste Ansatz letztendlich verworfen.
Letztendlich konnte uns völlig überraschend Professor Andreas Resch helfen, da er sich in seiner Freizeit selbst viel mit Hardware beschäftigt. Der jetzt implementierte Ansatz misst die Zeit zwischen den jeweiligen Flanken und berechnet sich daraus das Verhältnis. Aus Performancegründen wurde dieser vollständig im Emulator implementiert, was einige Änderungen im LED-Code notwendig machte.

Unklarheiten in der USB-Master-Dokumentation

Die USB-Master-Dokumentation war in vielen Fällen unklar oder nicht genau genug, so dass Annahmen getroffen wurden, welche im Punkt "Unterschiede zum USB-Master" dokumentiert sind.

Ein großes Problem gab es bei der Implementierung der `ORG`-Direktive. Diese setzt die Position im generierten Bitcode an eine bestimmte Position womit es möglich wird an beliebige Stellen (zum Beispiel `0x100` für die Interrupt-Tabelle) zu schreiben. Jedoch wurde der Befehl so interpretiert, dass es den Pointer immer um die gegebene Anzahl an Bytes relativ nach vorne rückt, nicht absolut setzt. Das Beheben dieses Fehlers machte es notwendig den Generator im Assembler komplett neu zu implementieren, was 6 Stunden Mehraufwand bedeutete.

Unterschiede zum USB-Master

Der originale USB-Master verhält sich in manchen Situationen nicht wie erwartet, zusätzlich sind manche Befehle oder Parameter nicht klar definiert, so dass Annahmen getroffen wurden.

END-Direktive

Beim USB-Master wird am Ende jedes Programmes ein `END`-Statement erwartet. Auf Grund eines Bugs müssen manchmal sogar mehrere hiervon verwendet werden, um das Programm starten zu können. In yasp ist das `END`-Statement optional.

Flags bei Word-Befehlen

Der USB-Master setzt bei diversen Word-Befehlen das Zero- und Carry-Flag nicht richtig. In yasp wurde dieses Fehlverhalten nicht übernommen.

DELAY und PAUSE

Beim USB-Master gibt es die Befehle `DELAY` und `PAUSE`, welche beide die Ausführung für eine gewisse Zeit anhalten. Es ist nicht klar definiert, welcher der Befehle ein Word-Register und welcher ein Word-Literal erwartet. In yasp können beide Befehle mit sowohl einem Register als auch einem Literal verwendet werden.

Beim USB-Master bedeutet `0` als Parameter die maximale Wartezeit. In yasp wird bei dem Wert `0` gar nicht gewartet.

Weiters wurde die Binärsignatur von `011XXXXX` (`X` = unbeachtet) zu `01100000` geändert, da es für unbeachtete Werte keinen technischen Grund gibt und ein fixer Wert leichter zu implementieren war.

Bezeichnung von Labels

In *yasp* ist es nicht möglich Labels mit Register-Bezeichnungen (z.B.: `w3004:` oder `b21:`) zu definieren, da dies zu Unklarheiten im Code führen kann.

WRITERAM und READRAM

Der USB-Master bietet keine Möglichkeit an beliebige Adressen im RAM zu schreiben oder diese zu lesen. Diese Funktionalität ist wichtig für Compiler welche den USB-Master oder yasp als Plattform verwenden. Dies wurde in yasp durch die zusätzlichen Befehle `WRRAM` und `RDRAM` ergänzt. Näheres ist in der Hilfe-Funktion zum jeweiligen Befehl zu finden.

Binärsignatur von LA

Beim USB-Master teilt sich `LA` die Signatur mit `MOV w,l`. Das ist mit dem modularen Aufbau von yasp nicht ohne Tricks realisierbar. Deswegen änderte sich die Binärsignatur zu

```
code     register   label-address
0x28     000ddddd   xxxxxxxx xxx00000
```

DEFINE-Direktive

Es wurde eine `DEFINE`-Direktive eingeführt, die es erlaubt, Synonyme im Sourcecode einzuführen. Eine solche wird zwar in der USB-Master-Dokumentation verwendet, allerdings nicht dokumentiert. Sie kann unter anderem verwendet werden, um den Code leichter lesbar zu machen, zum Beispiel:

```
DEFINE  LED_YELLOW 3
HIGH    LED_YELLOW
; --- vs.
HIGH    3
```

Umgesetzte Instruktionen

Wir haben knapp zwei Drittel der im USB-Master verfügbaren Instruktionen umgesetzt. In der nachfolgenden Tabelle ist sichtbar welche Befehle geplant und umgesetzt wurden.

Kategorie	Befehl	Geplant	Umgesetzt
basic	ADD	Ja	Ja
basic	SUB	Ja	Ja
basic	CMP	Ja	Ja
basic	OR	Ja	Ja
basic	AND	Ja	Ja
basic	XOR	Ja	Ja
basic	MOV	Ja	Ja
basic	MUL	Ja	Ja
basic	DIV	Ja	Ja
basic	INC	Ja	Ja
basic	DEC	Ja	Ja
basic	INV	Ja	Ja
basic	CLR	Ja	Ja
basic	RR	Ja	Ja
basic	RL	Ja	Ja
basic	DELAY	Ja	Ja
basic	RANDOM	Ja	Ja
ram	SETUP		
ram	STATE		
rom	READ		
rom	WRITE		
stack	PUSH	Ja	Ja
stack	POP	Ja	Ja
debug	DEBUG	Ja	Ja
rc5	RCFRE		
rc5	RCRECQ		
rc5	RCSEND		

Kategorie	Befehl	Geplant	Umgesetzt
serial	SEROUT		
serial	TXBAUD		
usb	USBREC		
adc	ADC0	Ja	Ja
adc	ADC1	Ja	Ja
adc	POT	Ja	Ja
gpio	HIGH	Ja	Ja
gpio	LOW	Ja	Ja
gpio	TOGGLE	Ja	Ja
gpio	PIN	Ja	Ja
M I/O	SOUND		
M I/O	PWMPRESC		
M I/O	MLRESET		
M I/O	BURST		
M I/O	COUNT		
M I/O	PULSIN		
M I/O	EDGE		
M I/O	PULSOUT		
M I/O	MLREAD		
M I/O	MLWRITE		
interrupts	ENABLE	Ja	Ja
interrupts	DISABLE	Ja	Ja
interrupts	RETI	Ja	Ja
debug	ECHO	Ja	Ja
debug	BREAK	Ja	Ja
interrupts	LA	Ja	Ja
directives	ORG	Ja	Ja

jmp	GOTO	Ja	Ja
jmp	CALL	Ja	Ja
jmp	RETURN	Ja	Ja
jmp	JZ	Ja	Ja
jmp	JNZ	Ja	Ja
jmp	JC	Ja	Ja
jmp	JNC	Ja	Ja
jmp	JMPI	Ja	Ja
M I/O	DIRS		
M I/O	PEEK		
M I/O	POKE		
M I/O	PWM0		
M I/O	PWM1		

directives	STRING	Ja	Ja
directives	DB	Ja	Ja
directives	DA	Ja	Ja
directives	DW	Ja	Ja
directives	END	Ja	Ja
serial	RXSTATE		
serial	RXINIT		
serial	RXLAST		
serial	RXFIRST		
serial	SERSTR		
serial	SERHEX		
serial	SERDEC		

Insgesamt wurden 47 von 79 Befehlen implementiert.

Anregungen für Folgeprojekte

Während der Entwicklung von *yasp* sind dem Projektteam mehrere Erweiterungsmöglichkeiten aufgefallen. Da die meisten hiervon nicht notwendig waren, wurden Sie nicht sofort implementiert, sondern wurden hier als Anregungen für Folgeprojekte aufgelistet.

Allgemeines

Compiler für eine Hochsprache

Ein größeres Projekt wäre ein Compiler, welcher mit yasp kompatiblen Assembler-Code erzeugt. Dieser könnte es ermöglichen, Hochsprachen wie C oder BASIC auf yasp auszuführen.

Mehrbenutzersystem

Ein Mehrbenutzersystem würde es mehreren Benutzern erlauben gleichzeitig an einem Dokument zu arbeiten, ähnlich wie Google Docs oder Etherpad-lite.

Mobile Endgeräte

Momentan werden mobile Endgeräte nur rudimentär unterstützt, da die Oberfläche nicht für Touch-Geräte optimiert ist. Zusätzlich müsste überprüft werden, ob die Rechenkraft eines Smartphones ausreichend ist um den Emulator auszuführen.

Debugger

Stepping

Aktuell werden die Step und Step Back Kommandos unterstützt. Weiter Step-Funktionen wie zum Beispiel Step out oder Step over wären sinnvoll. Step out führt das Programm bis zum nächsten `RET` aus. Step over kann nur bei einem `CALL`-Befehl verwendet werden. Es setzt einen Breakpoint auf die nächste Zeile nach dem `CALL`, so dass die Sub-Prozedur übersprungen wird.

Code Injection

Um neue Code-Schnipsel zu testen, könnte der Debugger dynamisch Code in den Emulator laden und diese Ausführen.

Passiv genutzte Register anzeigen

Da in den ersten 16 Word-Registern alle 32 Byte-Register beinhalten, wird beispielsweise bei der Verwendung des Registers `b0` auch `w0` verändert. Im Debugger sollten daher auch diese passiv genutzten Register angezeigt werden.

Editor

Speichern der Breakpoints

Breakpoints werden nicht mit der Datei abgespeichert, da diese normalerweise nur während einer "Debugging-Session" interessant sind. Trotzdem könnte man diese mit der Datei zusammen abspeichern, um zum Beispiel einem anderen Nutzer zu ermöglichen, dieselben Breakpoints zu verwenden.

Schnellhilfe für Direktiven

Die Direktiven (zum Beispiel `DW`, `STRING` oder `ORG`) sind zwar im "Tutorial" von yasp beschrieben, allerdings bietet der Editor selbst keine Hilfe dazu an.

Hardware

Sound

Der USB-Master unterstützt die Ausgabe von Tönen, entweder mittels PWM oder der fertigen `SOUND`-Instruktion. Dieses Feature könnte in einer späteren Version von yasp mittels der HTML5-Audio-API nachgebaut werden.

LED-Matrix

Als weiteres Hardware-Element war eine LED-Matrix angedacht, welche mit einem seriellen Protokoll angesprochen wird.

7-Segment-Display

In der 1. Klasse wird auch die Ansteuerung eines 7-Segment-Displays unterrichtet. Dieses könnte in yasp nach Implementierung der "Hardware mit mehreren Pins" realisiert werden, um noch mehr Ziele des Unterrichts abzudecken.

Hardware mit mehreren Pins

yasp unterstützt nur Hardware mit einem Ausgangs- bzw. Eingangspin, wie zum Beispiel LEDs oder Potentiometer. Für komplexere Hardwareelement wie ein 7-Segment-Display oder eine LED-Matrix ist es notwendig, mehrere Pins pro Element verwalten zu können.

Schrittmotor

Um ein weiteres Beispiel der Hardwareansteuerung im Unterricht durchnehmen zu können, wäre die Implementierung eines Schrittmotors in yasp möglich. Da ein Schrittmotor jedoch naturgemäß mehrere Eingänge hat, müsste zuerst das Hardware-System von yasp auf Elemente mit mehreren Pins erweitert werden.

Persönliche Stellungname

Michael Lutonsky

In der ersten Klasse meiner Schullaufbahn an der HTL Spengergasse, wurden meine Klassenkollegen und ich mit dem USB-Master und Assembler konfrontiert. Ich wusste zwar, was Assembler ist, habe die Sprache aber zu dem Zeitpunkt jedoch noch nie verwendet. Die ersten Übungen waren recht einfach, die späteren allerdings nicht, da man hier viel Zeit mit der Fehlersuche verbrachte. Der Debugger des echten USB-Masters hat leider nicht sehr viele Funktionen, was die Möglichkeiten zur Behebung eines Fehlers erheblich einschränkt. Das war einer der Punkte die mich immer am USB-Master gestört haben.

Einige Jahre später wurde uns in der vierten Klasse angeboten den USB-Master in den Browser zu verlegen. Nach kurzer Überlegung holte ich mir Robert Fischer mit ins Boot. Zusammen haben wir die aktuelle Situation analysiert und uns dann für das Projekt entschieden. Bei den Projektpräsentationen in der 4. Klasse fand sich kein weiteres Projektmitglied, weswegen es bei einer Teamgröße von zwei Leute blieb. Da wir beide viel Erfahrung im Bereich der Softwareentwicklung haben, schränkte dies den Umfang des Projekts aber nicht wirklich ein.

Ich habe bereits in der 3. Klasse zum Spaß einen Emulator für den USB-Master entwickelt, allerdings in C#.net, nicht in JavaScript. Diesen habe ich auch auf den „netduino“ geportet, sodass man am Ende wieder echte Hardware hatte. Hier waren allerdings alle Befehle fest in den Emulator einprogrammiert, weswegen die Programmstruktur sehr unterschiedlich zu yasp ist. In Zukunft könnte ich mir auch vorstellen den Emulator von yasp auf „tessl.io“ (eine Hardwareplattform auf welcher man JavaScript ausführen kann) zu portieren.

Wir beide haben jeweils die Programmteile erledigt in denen wir uns fachlich am sichersten waren. Robert Fischer hat hauptsächlich am Assembler gearbeitet, da er schon mehrere Compiler geschrieben hat und ich habe den Emulator entwickelt. Dies hat, bis auf die ORG-Direktive, zu einem sehr reibungslosen Projektablauf geführt.

Da war das gesamte Projekt in den Sommerferien durchgeplant haben, stellte sich die Kommunikation während des Projekts als sehr einfach heraus. Es war immer klar welche Arbeiten unter eine allgemeinen Bezeichnung wie „Debugger“ fallen und wie genau diese zu implementieren sind. Aus diesem Grund konnten wir die meisten Meetings über das Internet und in weniger als einer Stunde abhalten, was viel Zeit gespart hat. Außerdem wurden so viele Diskussionen über Implementationsdetails während des Projektes vermieden, da diese in den Sommerferien stattfanden.

Insgesamt bin ich mit dem Projekt sehr zufrieden. Wir haben alles erreicht, was wir erreichen wollten und konnten sogar noch ein paar KANN-Ziele unterbringen. Wenn sich Zeit findet werde ich auf jeden Fall noch ein paar der verbleibenden Punkte (siehe „Anregungen für Folgeprojekte“) in meiner Freizeit erledigen – einfach weil es mir bis jetzt sehr viel Spaß gemacht hat an yasp weiter zu Programmieren. Weiters hoffe ich, dass sich nächstes Jahr ein Diplom- oder Begleitprojekt findet, welches ebenfalls yasp ein wenig nach vorne treiben wird.

Robert Fischer

Aus der technischen Sicht war für mich yasp nichts Neues: Ich habe bereits zuvor ein Projekt entwickelt, das ähnlich wie unser aktuelles Projekt im Browser lief (mit Entwicklungsumgebung und Compiler), darum habe ich bereits Erfahrungen aus diesem Projekt von Anfang an in die Entwicklung einbringen können. Beispielsweise habe ich von Anfang an darauf bestanden, dass sowohl der Assembler, als auch der Emulator in einem WebWorker läuft (was sich als sehr wichtig herausgestellt hat). In Zusammenspiel mit den Erfahrungen von MIchael Lutonsky im Bereich der Webentwicklung und Emulation des USB-Masters, war die Umsetzung des Projektes sehr "straightforward". Bis auf kleinere Probleme mit bestimmten "Browsern" und Herausforderungen mit der Performance von JavaScript würde ich sagen lief alles weitgehend nach Plan.

Aus der Sicht des Projektmanagement, war unser Diplomprojekt für mich weitgehend Neuland: Ich habe zuvor noch nie SCRUM angewandt, geschweige denn von A bis Z umgesetzt. Es hat sich als schwieriger als gedacht herausgestellt, da die Sprints während der Schulzeit eben nicht immer die oberste Priorität hatten. Dennoch finde ich, eignet sich von den agilen Methoden SCRUM am besten, weil es sehr stark auf Kommunikation setzt.

Unser Team war mit zwei Personen sehr klein, was dazu führte, das Kommunikation, meiner Meinung nach, kein Problem war. Falls jemand ein Problem hatte oder es eine Frage gab, wurde sie sofort über Instant Messaging (Skype) gestellt und es wurde sofort beantwortet - informell und schnell. Ich finde, dass die Teamgröße und die Teammitglieder für dieses Projekt perfekt gewählt wurden. Michael Lutonsky war ein sehr fähiger Projektleiter und hat sich fast schon perfektionistisch für das Projekt eingesetzt.

Die Meilensteine die wir uns am Anfang des Projektes gesetzt haben, haben sich als gut geplant herausgestellt, was sich in der besonders frühen und vor allem erfolgreichen Projektabnahme (Ende Februar) erkennen lässt.

Die HTBLuVA Spengergasse, vertreten durch Dipl.- Ing. Jelinek war ein sehr angenehmer Projektpartner, da wir jederzeit Fragen stellen konnten und wir immer gute Antworten bekommen haben. Das einzige was ich zu bemängeln habe, war die Tatsache, dass uns 100,00€ Budget zugestanden wurde für Entwicklungsumgebungen und Domainkosten, die wir leider nicht erhalten haben, wodurch diese Kosten weitgehend auf unseren Rücken fiel.

In Zukunft könnte ich mir vorstellen weiter an yasp zu arbeiten und neue Funktionen und Verbesserungen zu implementieren. Das Projekt hat viel Potential und sollte dieses auch ausschöpfen!

Styleguides

Assembler-Styleguide

Assemblercode kann auf viele Arten formatiert werden. Um ein einheitliches Bild in dieser Dokumentation festzulegen wird, hier dieses Format definiert.

Grundsätzliche Regeln

- Einrückung mit TABs, empfohlene Tabweite sind 8 Spaces
- Alles bis auf Labels und Kommentare um mindestens eine Stufe einrücken
- Kommentare
  - müssen nach dem „;“ ein Leerzeichen haben
  - müssen falls in der selben Zeile ein Befehl ist eingerückt werden
  - Tab vor dem “;” wenn hinter einem Befehl
- Befehle
  - Leerzeichen nach dem Beistrich in der Parameterliste
  - immer komplett in Großbuchstaben
  - TAB nach dem Befehlsnamen
- Labels
  - CamelCase
  - Kein Befehl in der selben Zeile
  - Kein Kommentar in der selben Zeile
- Registernamen mit kleinen Buchstaben
- Immer ein `END`-Befehl am Ende des Programmes, welcher nicht eingerückt werden darf

Beispiel

```
loop:
       HIGH    3       ; rot
       DELAY   w1      ; bleibt lange rot
       MOV     w4, 4
       JMP     subOne

back:
       MOV     w4, 6
       JMP     loop    ; Endlosschleife

subOne:
       MOV     w4, 5
       JMP     back
END
```

JavaScript-Styleguide

JavaScript ist die primäre Programmiersprache in der *yasp* entwickelt wurde. Um eine klare Struktur festzulegen wurden folgende Regeln definiert.

Grundsätzliche Regeln

- Jede Zeile muss mit einem Semikolon abgeschlossen werden
- Kommentare müssen ein Leerzeichen zwischen Text und Bezeichner haben
- Funktionen sollten wenn möglich als JSDoc dokumentiert werden
- Vor und nach Operatoren muss ein Leerzeichen sein
- Nach Schlüsselwörtern (`if`, `while`, `for`,...) muss ein Leerzeichen sein
- Innerhalb leerer Klammern („`{}`“, „`[]`“, „`()`“) muss ein Leerzeichen eingefügt werden
- Der Code sollte eingerückt werden, wobei jede öffnende geschwungene Klammer eine neue Ebene startet und jede schließende geschwungene Klammer eine Ebene schließt
- Eingerückt wird mit 2 Leerzeichen

Beispiel

```
// comment
/*
 * comment 2
 */
exports.externalfoo = function (a, b, c) {
  var foo = 42;
  var bar = "foo";
  var fuz = null;

  foo = {
    key: "value",
    key2: 42,
    foo: { },
    example: [ ]
  };
  // one line per IF/WHILE/DO/FOR, not 2, not 3. ONE.
  if (true) {
      // ...
  }
  if (a == 42) {
    // foo
  }
```

```
  for (var i = 0; i < 42; i++) {
    // bla
  }
  do {
    // code
  } while (false);
  internalfoo(b,
    function (err, val) {
      val *= 42;
      console.log(val);
    }
  );
};
```

HTML-Styleguide

HTML wird zur Strukturierung der Daten im Browser verwendet.

Grundsätzliche Regeln

- Jeder HTML Tag startet eine neue Ebene und der dazugehörige End-Tag beendet die Ebene
- Eingerückt wird mit zwei Leerzeichen
- HTML Tags werden klein geschrieben
- Es wird kein Inline-CSS oder Inline-JavaScript verwendet
- Als Doctype wird das HTML5 doctype verwendet

Beispiel

```
<!doctype html>
<head>
  <title></title>
</head>
<body>
  <h1>Hello World</h1>
  <p class="foo">
    foobar
  </p>
</body>
</html>
```

CSS-Styleguide

CSS wird zur Formatierung und Darstellungssteuerung in *yasp* verwendet.

Grundsätzliche Regeln

- Jede offene geschwungene Klammer startet eine neue Einrückungsebene und schließende geschwungene Klammer schließt eine Einrückungsebene
- Eingerückt wird mit zwei Leerzeichen
- Nach jedem Doppelpunkt muss ein Leerzeichen kommen, davor keines
- Nach und vor jedem Verknüpfungsoperator muss ein Leerzeichen kommen

Beispiel

```
#foo {
  height: 100px;
  width: 20px;
}

#foo,
#bar {
  color: red;
}

ul#asd > li.bar {
  background-color: red;
}
```

JSON-Styleguide

Die JavaScript Object Notation dient zur Datenhaltung und Kommunikation zwischen den einzelnen Komponenten.

Grundsätzliche Regeln

- Es gelten grundsätzlich die selben Regeln wie für JavaScript
- Zusätzlich muss jeder Key in doppelten Anführungszeichen („“) maskiert werden, sobald es ein Key-Value Paar ist

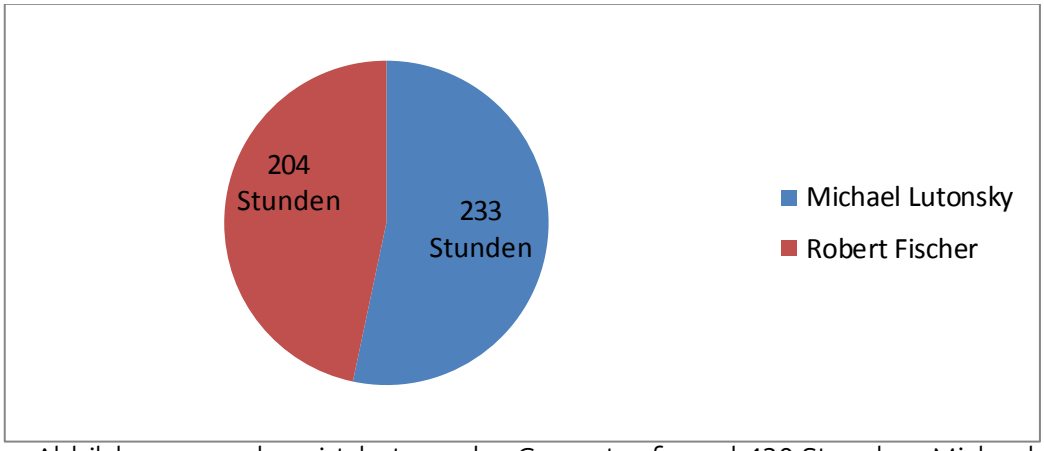
Beispiel

```
{
  "key": {
    "subkey1": [
      42,
      43,
      {
        "42": 42
      }
    ],
    "subkey2": {
      "luto": 42,
      "fisch": "foo",
      "asdf": { },
      "lol": [ ]
    }
  }
}
```

Projektdokumentation

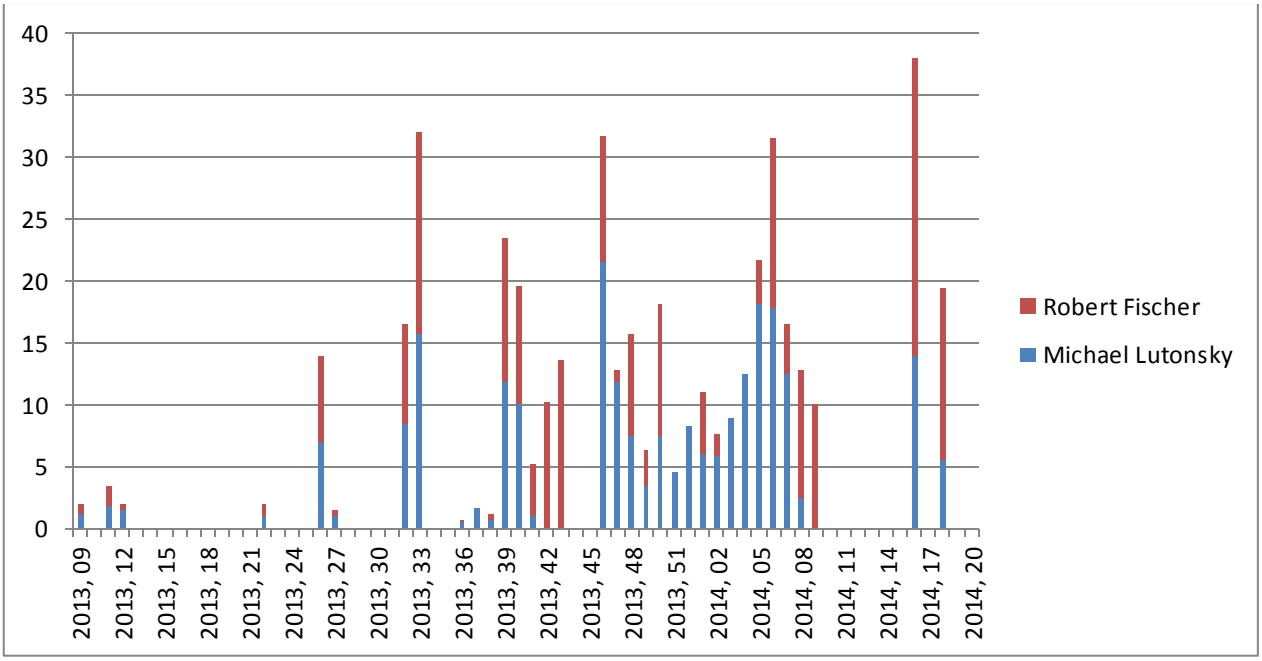
Zeitaufwandsauswertung

Teammitglied



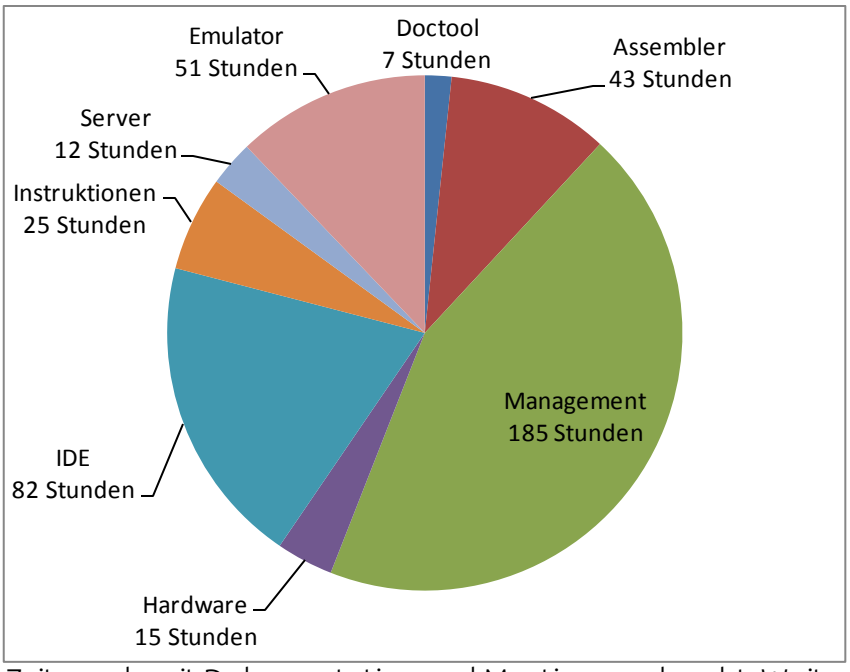
Wie in der Abbildung zu sehen ist betrug der Gesamtaufwand 430 Stunden. Michael Lutonsky investierte 30 Stunden mehr was mit einem geringeren schulischen Aufwand begründet wird. Insgesamt haben wir rund 30 Stunden mehr investiert als gedacht, was vor allem auf die Probleme mit PWM und dem Firefox-Browser zurückzuführen ist. Dies hatte allerdings keine Auswirkungen auf die Projektdauer. Die Arbeit wurde dennoch rechtzeitig fertiggestellt.

Zeitabschnitt



Woche 9, 2013 bis 31, 2013 bestand aus der Vorprojektphase. Hier wurden erste Meetings getätigt und das Pflichtenheft verfasst. Direkt danach wurde mit der eigentlichen Implementierung begonnen, welche in Woche 43 auf Grund des Ausfalls von Michael Lutonsky (siehe Sprint-Protokolle) stagnierte. Der Großteil der Entwicklung fand in den Monaten um den Jahreswechsel statt und war mit der 9. Woche abgeschlossen. In die Abschlussdokumentation wurde vor allen von Robert Fischer investiert, da er noch Stunden aufzuholen hatte.

Programmteil



Ein Großteil der Zeit wurde mit Dokumentation und Meetings verbracht. Weitere Schwerpunkte waren wie erwartet die IDE, der Emulator und der Assembler, wobei sich der Aufwand des Debuggers auf IDE und Emulator aufgeteilt. Auf Grund eines verbesserten Test-Systems konnte die benötigte Zeit zur Implementierung der Instruktionen drastisch reduziert werden. Bis auf den Assembler und die Serverkomponente wurde an allen Programmteilen von beiden Mitgliedern gearbeitet.

Die ungefähre Aufteilung nach Schwerpunkten sieht wie folgt aus.

Programmteil	Michael Lutonsky	Robert Fischer
Emulator	X	
Doctool		X
Assembler		X
IDE – Debugger	X	
IDE – Editor		X
Instruktionen	X	
Server	X	

Milestoneplanung

In der folgenden Tabelle sind unsere Milestones und die zugehörigen Daten aufgeführt. Es wurden alle Milestones bis auf einen rechtzeitig fertiggestellt. Einzig der 2. Milestone wurde hat sich um ein Monat verzögert, was an der Schnittstelle zwischen Filemanager und Serverkomponente lag. Dieser Unterpunkt was unabhängig von anderen Milestones, so dass die Entwicklung trotzdem wie geplant fortgeführt werden konnte.

Name	Beschreibung	Ziel Datum	Ist Datum
• Assembler • Emulator	Fertigstellung des Assemblers und Emulators inklusive aller Befehle (Emulator ohne Debugging)	Anfang Dezember	20. Dezember
• IDE • Filemanager • Serverkomponente	Fertigstellung der jeweiligen Use Cases Server samt REST API	Mitte Jänner	19. Februar
• Debugger	Debugger Oberfläche und Funktionen im Emulator	Ende Jänner	18. Jänner
• Endnutzerdokumentation • Befehlsdokumentation • Abnahme		Ende Februar	20. Februar

Sprint Protokolle

1. Scrum Meeting (0.1)

Datum	27. September 2013
Beginn	08:00
Ende	09:40
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/

Erkenntnisse und Aktivitäten	**Nächster Sprint**
	• Vollständige Implementierung des MOV-Befehls samt Emulatorarchitektur
	• Implementierung des Assemblers
	• Implementierung aller Direktiven
	
	Andere Aktivitäten
	• Hinzufügen von User Storys in
	  • API
	  • Debugger
	  • Editor
	• Anpassungen im Pflichtenhefts
	• yasp git Repo erweitert

Ergebnis	Wir können unseren ersten Sprint starten.

Meeting mit Professor Jelinek

Datum	3. Oktober 2013
Beginn	16:00
Ende	16:30, 30 Minuten
Teilnehmer	Robert Fischer, Michael Lutonsky, Prof. Robert Jelinek
Typ	Außerordentlich
Grund wenn außerordentlich	Bericht der aktuellen Vorgänge
Erkenntnisse und Aktivitäten	**Repl** Das REPL[1] mit samt der MOV-Befehle wurde präsentiert. Es wurde ein geringfügiges Problem am Assembler entdeckt, welches noch zu evaluieren ist: „`MOV b0, ab`“ wird vom Assembler als Syntaxfehler gesehen, jedoch könnte der User erwarten dass dies als Hexa-Dezimale Zahl interpretiert wird. **Dokumentation und Milestones** Die vorläufigen Milestones[2] wurden besprochen. Dabei wurde festgestellt, dass der Debugger (samt Register-Ansicht) vom Editor und der IDE unabhängig und erst später entwickelt wird. Des Weiteren wurde angemerkt, dass es kein Problem ist, wenn das Projekt früher fertig wird, solange das Stundenbudget ausgenutzt wird. **Tessl.io** Prof. Jelinek hatte noch keine Zeit sich tessl.io anzuschauen. Verschoben auf das nächste Meeting. [1]: http://yasp.me/yasp/src/app/test/repl.html [2]: http://pad.yasp.spengr.co/p/milestones
Ergebnis	• Alles klar für den nächsten Sprint. • Es wurde vergessen zu fragen, wie es bezüglich Lizenz aussieht. Dies muss beim nächsten Meeting nachgeholt werden. • Tessl.io muss bei einem späteren Meeting besprochen werden

2. Scrum Meeting (0.2)

Datum	4. Oktober 2013
Beginn	08:00
Ende	08:40, 40 Minuten
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**Vorhergehender Sprint** **Aufgaben** • Assembler Implementierung erfolgte wie geplant • Alle Testcases werden erfolgreich ausgeführt • Implementierung einer (ungeplanten) interaktiven Assembler Umgebung (REPL), da man damit leichter den Fortschritt herzeigen kann **Probleme** • Dekodieren der Befehle war mehr Aufwand als geplant (5 Stunden statt 1 Stunde). Dies war allerdings kein größeres Problem. **Nächster Sprint** • Implementierung aller Basic-Befehle • Implementierung der Hardwareelemente
Ergebnis	Wir können unseren zweiten Sprint starten.

3. Scrum Meeting – Pause

Datum	15. November 2013
Beginn	14:15
Ende und Dauer	18:30, 75 Minuten
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Außerordentlich
Grund wenn außerordentlich	Pause beendet, weitere Pläne und Milestonebesprechung
Erkenntnisse und Aktivitäten	**Pause** Herr Lutonsky hat mit 14. Oktober wegen eines anderen Projektes eine Pause angetreten. Diese ist seit dem 10. November beendet und der normale Projektbetrieb ist wieder hergestellt. In dieser Pause wurde von Herrn Fischer die Arbeit an der IDE begonnen. Michael Lutonsky hat bereits einen Teil der fehlenden Stunden aufgeholt und wird sich weiterhin bemühen eine Stundengleichheit wiederherzustellen. Aufgrund dieser Vorkommnisse haben wir das Sitzungsprotokoll nicht einhalten können und versuchen unseren regulären Arbeitsablauf nun wiederherzustellen. **Vorstudien-Präsentation** Die Präsentation wurde für den 27. November vorbereitet. **Milestones** Die Milestones sollten trotz der Pause eingehalten werden können. Allerdings muss in den verbleibenden Wochen zur Erreichung des ersten Milestones *Fertigstellung des Assemblers inklusive aller Befehle* und *Fertigstellung des Assemblers inklusive aller Befehle ohne Debugging-Features* mehr gearbeitet werden. **Optimierung des Emulators** Der Emulator wurde von 17 Hz auf 22 kHz optimiert. **Neues Test-System** Das Testsystem wurde optimiert, so dass das Schreiben eines Tests nun durchschnittlich 2 Minuten, statt von 5-7 Minuten dauert.

	Dokumentationsformat
	Aktuelles Format:
	`{ "name": "MOV", "description": "moves stuff" }`
	Neues Format:
	`{ "name": "MOV", "description": [ "en": "moves stuff", "de": "schiebt zeug" ] }`
	Zusätzliche Änderungen:
	Wenn die description `null` ist, wird die Beschreibung von einem Befehl übernommen, welcher den selben Namen hat.

Ergebnis	Siehe Erkenntnisse und Aktivitäten.

4. Scrum Meeting (0.3)

Datum	13. Dezember 2013
Beginn	9:00
Ende und Dauer	9:30 , 30 Minuten
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**Aufholsprint I** Implementierung aller verbleibenden Befehle bis auf • Interrupts • Analoge HW-Befehle • RANDOM • PAUSE Zusätzlich werden im Emulator nicht immer alle Parameter geladen, so ist zum Beispiel bei `MOV w0,w1` der Wert von `w0` irrelevant. **ORG-Direktive** Während der Programmierung der Interrupts sind wir auf ein Problem mit der Assembler Direktive `ORG` gestoßen. Diese hat den Maschinencode falsch generiert, was dazu führte dass der Generator neuimplementiert werden musste. Diese Aufgabe hat Robert Fischer erledigt. **Assembler Fehlermeldungen** Die Fehlermeldungen des Assemblers bei invalider Syntaxeingabe sind nun um einiges verständlicher, da der Assembler nun mehr Fehler einzeln abprüft.

5. Scrum Meeting (0.4)

Datum	20. Dezember 2013
Beginn	9:00
Ende und Dauer	9:30, 30 Minuten
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**Aufholsprint II** Implementierung der verbleibenden Emulator-Features: • Interrupts • Analoge HW-Befehle • RANDOM • PAUSE **Präsentationsvorbereitung** Der REPL wurde für die Statuspräsentation um eine schönere Ausgabe ergänzt. Die IDE wurde ebenfalls aufgeräumt, so dass eine reibungslose Präsentation sichergestellt werden konnte. **Assembler Fehlermeldung** Wenn im Assembler ein Befehlsname als Label verwendet wurde (zum Beispiel `SUB: GOTO SUB`), dann führte dies zu einer unverständlichen Meldung, die vom Benutzer nur schwer interpretierbar war.
Ergebnis	Der Emulator ist nun Feature-Complete Präsentationsdemos vorbeitetet Verbesserung der Fehlermeldungen des Assemblers Milestone 1: *Assembler und Emulator fertiggestellt*

6. Scrum Meeting (0.5)

Datum	27. Dezember 2013
Beginn	17:00
Ende und Dauer	17:30, 30 Minuten
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/

Erkenntnisse und Aktivitäten	**Editorintegration**
	Assembler und Emulator wurden in den Editor integriert. Es ist jetzt möglich alle Features der beiden im Editor zu verwenden. Zusätzlich werden unter dem Textfeld weitere Informationen und die Parameter des aktuellen Befehls angezeigt.
	
	Hardwareintegration
	Im Debugger wird nun die Hardware angezeigt und es ist möglich diese mit Befehlen wie z.B. `HIGH` vom Emulator anzusprechen.
	
	Hardwareanpassungen
	Die Wertebereich des Potis wurde von 0-359 zu 0-255 geändert. Die Werte des Pushbuttons wurden von `true` und `false` auf `0` und `1` geändert.
	
	Bugfixes im Assembler
	Der Assembler warf eine Syntaxfehlermeldung, wenn ein Kommentar in einer Zeile war, wo bereits ein Befehl vorhanden war.
	`MOV b0, 10 ; Ein super Kommentar`
	Weiters war der Befehl `DA` nicht korrekt implementiert, so dass er eine falsche Adresse in den Maschinencode geschrieben hat.
	`ORG 1meinlabel: DA meinlabel ; hat die falsche Adresse in den Maschinencode geschrieben`

7. Scrum Meeting (0.6)

Datum	07. Jänner 2014
Beginn	18:00
Ende und Dauer	19:00, 1 Stunde
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**FileManager** Der FileManager wurde implementiert, das heißt, man kann nun einfach Dateien speichern, laden, löschen und umbenennen. Dabei wurde ein Driver System implementiert, wodurch es möglich ist, den FileManager sowohl lokal als auch am Server speichern zu lassen. **Server** Die Serverkomponente von yasp ermöglicht es, in Verbindung mit dem Filemanager Dateien auf den Schulservern zu speichern. Hierbei loggt sich der User mit dem Active Directory Account ein und die Dateien werden entsprechend gespeichert. Der Server wurde in PHP geschrieben. **Localization** Localization ist die Übersetzung des Programmes in verschiedene Sprachen. Hierbei haben wir eine eigene Library geschrieben, die dynamisch den HTML DOM abarbeitet und jegliche Texte in die entsprechende Sprache übersetzt. Die Daten für die Sprachen sind in den Dateien `langs/de.js` für Deutsch und `langs/en.js` für Englisch hinterlegt.

8. Scrum Meeting (0.7)

Datum	13. Jänner 2014
Beginn	18:00
Ende und Dauer	19:00, 1 Stunde
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**Debugger** In diesem Sprint wurden die Grundfunktionen des Debuggers implementiert. Ein Großteil der Arbeit war die Schaffung der notwendigen Infrastruktur, wie zum Beispiel das Tab-System oder die eigentliche Verbindung zum Emulator selbst. Zusätzlich kann sich der Nutzer nun den Inhalt des RAMs und ROMs anschauen. Der Debugger beherrscht auch alle grundlegenden Funktionen wie Stepping oder das Anzeigen der aktuellen Zeile im Editor. **Debug-Commandos** Die beiden Debug-Instruktionen `ECHO` (eine Zeichenkette ausgeben) und `DEBUG` (einen Registerwert ausgeben) wurden implementiert und die notwendige Ausgabemöglichkeit im Debugger geschaffen. Somit ist es nun möglich Werte zum Debugger zu schicken. **Technische Dokumentation** Erste Schritte im Bereich der technischen Dokumentation wurden ebenfalls getätigt. Es gibt nun eine genaue Beschreibung des Instruktionsformates von yasp und grundlegende Informationen zum Emulator.

9. Scrum Meeting (0.8)

Datum	18. Jänner 2014
Beginn	18:00
Ende und Dauer	19:00, 1 Stunde
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**Debugger** Im letzten Sprint wurden weitere Debugger Features implementiert. Das Wichtigste ist wohl der Register-Tab, in dem man die Werte alle im Code verwendeten Register überprüfen kann. Dort ist auch eine History aller vorherigen Werte sichtbar, in der jede Änderung eines Wertes farbig markiert wird. In diesem Tab wird auch der aktuelle Wert der SPRs (Special Purpose Registers) wie SP oder PC sowie der Stack selbst angezeigt. Der Debugger speichert bei jedem Step den aktuellen Zustand des Emulators ab. Diese gespeicherten Zustände können mit dem "Step Back" Befehl wieder hergestellt werden. Somit ist es möglich, in der Ausführung wieder einen Schritt zurück zu gehen, was beim Debuggen sehr hilfreich sein kann. **PushButton sperren** Beim Debuggen ist es manchmal notwendig, einen der beiden Buttons zu drücken und gleichzeitig zu Steppen. Dies war bislang nicht möglich, da ein Button nur solang gedruckt blieb, wie die Maustaste selbst. Es ist jetzt möglich mit einem SHIFT-Klick die Status des Buttons zu "sperren" womit er auch gedrückt bleibt wenn die Maus den Button verlässt.

10. Scrum Meeting (0.9)

Datum	08. Februar 2014
Beginn	18:00
Ende und Dauer	19:00, 1 Stunde
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**Firefox-Probleme** **Webworker-Messages** Firefox hat im Gegensatz zu anderen Browsers Probleme mit einer großen Anzahl von Webworker-Messages, welche zum Beispiel durch das schnelle Blinken einer LED zustanden kommen können. Es war somit möglich den gesamten Browser mit einem yasp-Programm zum Stillstand zu bringen. In den betreffenden Funktionen im Emulator wurde Rate-Limiting (das Limitieren der maximalen Anzahl von Nachrichten in einer bestimmten Zeit) eingesetzt um dieses Problem zu mildern. **SetTimeout bzw. SetInterval** Da JavaScript immer nur einen Thread hat und der Emulator quasi gleichzeitig Befehle ausführen und Nachrichten von dem Debugger empfangen muss wird für die Befehle die `setTimeout` bzw. `setInterval`-Funktion verwendet. Diese funktioniert im Firefox nach einer bestimmten Anzahl von Aufrufen schlicht nicht mehr, was zu massiven Problemen im Emulator geführt hat. Der zugehörige Bug-Report bei Mozilla findet sich hier: https://bugzilla.mozilla.org/show_bug.cgi?id=972829 **Layout** Es wurden diverse Anpassungen im Layout des Debuggers vorgenommen, da manche CSS-Funktionen im Firefox unerwartet langsam sind, was der Gesamtperformance sehr geschadet hat. **Breakpoints** Sowohl im Emulator aus auch im Debugger wurden Breakpoints implementiert. Es ist somit möglich das Programm an bestimmten Stellen anzuhalten um einen Fehler schneller zu finden.

11. Scrum Meeting (1.0)

Datum	19. Februar 2014
Beginn	18:00
Ende und Dauer	19:00, 1 Stunde
Teilnehmer	Michael Lutonsky, Robert Fischer
Typ	Wöchentlich
Grund wenn außerordentlich	/
Erkenntnisse und Aktivitäten	**In Zeile Springen** Diese Funktion ermöglicht es im Debugger in beliebigen Codezeilen zu springen. Hierzu drückt man den Button `Jump To` und dann in die designierte Zeile und der Emulator setzt den Program Counter auf die ausgewählte Zeile. **Serverkommunikation** Der Driver für die Serverkommunikation zwischen Server und FileManager wurde implementiert, so dass es nun im Browser möglich ist auch auf den Server zu speichern. **Kontextsensitive Autovervollständigung** Die Autovervollständigung in der IDE zeigt nun nicht mehr alle Kommandos und mögliche Register an, sondern basierend auf der aktuellen Cursorposition (Kontextsensitiv). Wenn der Cursor nun beim Befehl `MOV` ist, werden nur Byte und Word Register angezeigt, da dies die einzigen validen Parameter für `MOV` an erster Stelle sind. **Schnellhilfe Verbesserungen** Die Schnellhilfe wird nun angezeigt, auch wenn der Befehl nicht fertig eingegeben wurde, sondern nur teilweise. Wenn der angegebene Befehl nicht eindeutig zuordenbar ist, so wird eine Liste möglicher Befehle angezeigt. Um dieses Feature implementieren zu können, musste der Assembler in der Lage sein syntaktisch invaliden Code dennoch in einen AST zu konvertieren. Dies wurde mit sog. UNKNOWN_COMMANDS AST Nodes erreicht.

Mehrere Parameter

Die Befehle `DB`, `DA`, `DW` und `STRING` unterstützen nun beliebig viele Parameter, wodurch es möglich ist mehrere Werte auf einmal in den Maschinencode zu schreiben.

Assembler GitHub Dokumentation

Die Dokumentation des Assemblers ist nun auf GitHub dokumentiert und für jeden einsehbar.

Rescale Breadboard

Wenn der Debugger skaliert wird, so wird auch das Breadboard samt Hardware Elemente an die neue Größe angepasst. Hierbei gab es kleinere Probleme mit dem Potentiometer, weil dieser fixe Pixelwerte verwendet hat.

Unit Tests in Befehlsdateien ausgelagert

Die Unit-Tests der Instruktionen des Emulators waren vorher dezentral im `tests/` Ordner gespeichert. Jetzt werden sie stattdessen direkt in der Befehlsdatei mitsamt der Implementation und Metadaten persistiert, wodurch es übersichtlicher wird.

Diverse Bugfixes

Mehrere kleinere Fehler in der IDE, Debugger und Emulator wurden behoben, wie zum Beispiel dass Strings rot markiert waren, die automatische Speicherfunktion nun fehlerfrei funktioniert und die Tabgröße wird nun bei Änderungen korrekt dargestellt.

Technische Dokumentation

Inhaltsverzeichnis

yasp-Assembler Tutorial ... 137
Processor Features ... 137
Commands ... 137
Sleep ... 138
Debugging ... 138
Jumping ... 138
Comparisons ... 138
IO ... 139
Stack ... 139
Subroutines ... 140
PWM ... 140
Interrupts ... 140
Assembler-Directives ... 141

Localisation ... 142
DOM ... 142
JavaScript ... 142

Instructions ... 143
Format of an instruction file ... 143
exec-function ... 145
Boilerplate ... 146

Hardware ... 147
Types ... 147
Boilerplate ... 147

FileDialog ... 148
Usage ... 148
Driver ... 148
Structure ... 148
File ... 149

Communicator ... 150
Specific communicator-implementations ... 150
Messages ... 150
API ... 150

Breadboards ... 152
Breadboard format ... 152
Boilerplate ... 154

Assembler ... 155
Instruction ... 155
Label ... 155
Multi-Pass Architecture ... 156
EBNF ... 157

Testing ... 159
Commands ... 159
Process ... 159

setup 159
steps 159
Format of literal values 160
Full example 160

Hardware 161
LED 161
PushButton 161
Potentiometer (aka Poti) 161

Emulator 162
Ticks 162
PWM 162
Stack 164
Data 164
Messages 164

Emulator Data 165
Data 165
State 165

Emulator Messages 168
Broadcasts 168
Messages 169

Testprogramme 171
1. Testprogramm 172
2. Testprogramm 173
3. Testprogramm 175
4. Testprogramm 176
5. Testprogramm 177
6. Testprogramm 178
7. Testprogramm 179
8. Testprogramm 179
9. Testprogramm 180
10. Testprogramm 181

yasp-Assembler Tutorial

yasp uses a custom assembler which was created a decade ago by a teacher of our school, HTL Spengergasse. It has been designed to help students understand computers better and is therefore really simple and easy to learn.
A number of documented sample programs can be found in `sampleprograms/.`
Basic Syntax

```
label:
        COMMAND parameter1, parameter2
        COMMAND parameter1
        ; comment
```

A valid label must:

- only contain Numbers, Letters, _ or –
- not be a command-name (e.g. `MOV`)

Processor Features

- 32 byte registers: `b0` - `b31`
- 32 word registers: `w0` - `w31`
- the content of the first 16 word-registers is equal to the byte-registers
- a number of IO-Pins, see IO
- Hardware-Interrupts
- Carry and Zero-Flag
- 16-byte stack
- Simple subroutines
- PWM-Simulation (=> soft blinking LEDs)

Commands

This is only a list of the most important commands or instructions.
Use the help-feature of the IDE to get a full list of commands and their parameters and descriptions.

Basic Commands

All of these commands take two parameters:

- `register`, `register`
- `register`, `literal`

The result of the operation will be written into the first parameter, the register.

- `MOV`
- `ADD` / `SUB`
- `DIV` / `MUL`
- `OR`, `XOR`, `AND`

```
MOV b0, 2
MOV b1, 3

ADD b0, b1 ; b0 is 5dec

MOV b3, 1

AND b3, b0 ; b3 is 4dec
```

```
; b0:     5dec = 00000101b
; b3:     1dec = 00000001b
; result: 1dec = 00000001b
```

Sleep

DELAY can be used to wait for a certain time. Both literal and word-registers can be used as the parameter. A value of 60000 roughly equals 900 milliseconds.

Debugging

There are a few commands which can help you on debugging.

- DEBUG sends the value of a register to the debugger
- PRINT sends a string to the debugger

```
MOV b0, 42

DEBUG b0  ; send the value of b0 (42) to the debugger
PRINT str ; print a string on the debugger

str:
        STRING "hello tutorial!"
```

Jumping

There instructions are used to control the program-flow.

- JMP label
- JC label / JNC label jump if the carry flag is (not) set
- JZ label / JNZ label jump if the zero flag is (not) set

```
main:
        INC b0   ; increment b0
        DEBUG b0
        JMP main
```

Comparisons

You can use the CMP-Command to compare two values. If the values equal the zero-flag will be set, if the 2nd value is bigger than the first value, the carry flag will be set. Using CMP and JC one can build simple loops:

```
       MOV b0, 5   ; max loop count
       MOV b1, 0   ; counter

loop:
       PRINT str1
       INC b1      ; increment the couter
       CMP b0, b1  ; compare couter and max
       JZ endLoop  ; jump if they equal
       JMP loop    ; otherwise continue looping

endLoop:
       PRINT str2

trap:
       JMP trap ; loop infinitly
```

```
str1:
        STRING "loooping"
str2:
        STRING "done loooping"
```

IO

There are a number of commands which can be used to control hardware connected to the emulator.

- HIGH / LOW turn a pin on or off
- TOGGLE invert a output pin
- PIN read binary value from pin
- ADC read analogue value of the ADC-Pins (legacy commands: ADC0, ADC1, ADC2)

Toggling LEDs

```
main:
        TOGGLE 3    ; switch LED on output-pin 3
        DELAY 30000 ; wait for ~450ms
        JMP main
```

Reading pushbutton

```
main:
        PIN 1 ; read value from pushbutton
        ; PIN will set the zero-flag to 1 if the pins state is zero, so:
        ;  pin is on  => z = 0 => JNZ jumps
        ;  pin is off => z = 1 => JZ  jumps
        JNZ on
        JMP main

on:
        TOGGLE 3  ; toggle an LED each time a button is pressed
        JMP main
```

Stack

There is a 16-byte-stack which can be accessed by the fairly standard PUSH and POP-instructions.

```
MOV b0, 0x20
MOV b1, 0x40
MOV b2, 0x60
; w0 = 0x2040
; w1 = 0x6000
; initial sp = 0x50


; b1 = 0x40
PUSH b1
; Stack = 0x40
; sp    = 0x51

; b0 = 0x20
PUSH b0
; Stack = 0x40 0x20
; sp    = 0x52
```

```
; w1 = 0x6000
PUSH w1
; Stack = 0x40 0x20 0x00 0x60
; sp    = 0x54

POP w0
; Stack = 0x40 0x20
; sp    = 0x52
; w0    = 0x6000
```

Subroutines

Subroutinenesting is also supported using the stack.

```
main
        PRINT str1
        CALL subRout
        PRINT str2

        JMP main

subRout:
        PRINT str3
        RETURN

str1:
        STRING "main1"
str2:
        STRING "main2"
str3:
        STRING "sub"
```

PWM

yasp supports the emulation of PWM-Outputs. This can be archived by switching an output-pin on and off rapidly. More information about the internal process can be found in the Emulator Documentation.
This code causes the LED to be dimmed:

```
pwm:
        HIGH 3
        PAUSE 3000
        LOW 3
        PAUSE 2000
        JMP pwm
```

Interrupts

The emulator supports hardware-interrupts. To make them work you need two things:

- `ENABLE` enables interrupts based on the given bitmask
- `RETI` jumps out of an interrupt
- Interrupt-table
    - starts at `0x100` * `0x100` => jump-adress for pin 0
    - `0x102` => jump-adress for pin 1
    - `0x104` => jump-adress for pin 2
    - ...

```
; Interrupt-Mask:
; 7 6 5 4 3 2 1 0
; 0 0 0 0 0 1 1 0b = 3dec

ENABLE 3

main:
      TOGGLE 5
      JMP main

isr1:
      TOGGLE 3
      RETI

isr2:
      TOGGLE 4
      RETI


ORG 0x100 ; set the current position in the bitcode to 0x100
DW 0      ; skip pin 0
DA isr1   ; interrupt service routine for pin 1
DA isr2   ; interrupt service routine for pin 2
```

Assembler-Directives

- `DW` write a literal word into the bitcode (length: 2 bytes)
- `DB` write a literal byte into the bitcode (length: 1 byte)
- `DA` write a literal label-adress into the bitcode (length: 2 bytes)
- `ORG` set the current position in the bitcode (see Interrupts)
- `STRING` write a zero-terminated string into the bitcode (see Debugging)
- `DEFINE` defines a assemble-time constant which will be replaced before the actual assembling process

Localisation

yasp uses a custom library for localisation. Strings can either be referenced in the DOM via data-attributes or in javascript code. The translations themselves live in `/src/app/langs/` and are represented by a simple key-value-format. The code for the translation-library itself is in `/src/app/js/l10n.js`.

DOM

A very simple example is getting a string and replacing the `innerText` of an element with the translation.

```
<h6 data-l10n="editor.about.version"></h6>
```

Additionally parameters can be used

```
<option value="2" data-l10n="settings.tabSize.spaces" data-l10n-p0="2"></option
>
```

Corresponding line in the language-file: `"settings.tabSize.spaces": "{0} spaces`
If you need to set HTML instead of the text add a `data-l10n-html="true"`-attribute.

JavaScript

Retranslate the document

```
yasp.l10n.translateDocument();
```

Get the currently active language

```
var lang = yasp.l10n.getLangName();
```

Get a translation using the key, the last parameter is optional.

```
var trans = yasp.l10n.getTranslation("settings.tabSize.spaces", ["2"]);
```

Corresponding line in the language-file: `"settings.tabSize.spaces": "{0} spaces`

Instructions

An instruction in yasp consists of a single file inside the `/src/app/instructions/` directory. There is no code in the emulator or assembler dealing with specific instructions. Although some new instructions might need additional features in the emulator.
The instructions in the `instructions`-directory are later combined into `/src/app/js/commands.js`. This is done by the grunt-task `commands` or `watchcommands`, which can be used to update the file automatically once something has changed. Internally this is done by the doctool.

Format of an instruction file

```
{
  // name of the instruction, used by the assembler to identify it in the code
  "name": "ADD",

  // documentation section
  // each instruction should be documented in English and German, but the Engli
sh version is more important.
  "doc": {
    "en": {
      // general description of what this command does
      "description": "Adds the values of both registers.",

      // if the command sets any flags you should document *when* these flags a
re set here
      // if there are no flags set at all you can ombit the "flags"-section tot
ally.
      "flags": {
        "z": "is set if the result is 0",
        "c": "is set if the result is greater than 255 (one byte)"
      }
    },
    "de": { /* same as "en"-version */ }
  },

  // Tests for this command, which are executed by the EmulatorTester. See /doc
/testing/instructions.md for more information.
  // A testcase here must not use any commands but its own, as this is the only
one available. Assembler directives may
  // be used, though.
  "tests": [ ],

  // the code is the static part of the bitcode which is representing this comm
and. It is at the very
  // beginning of the bitcode and must be unique across all instructions since
the emulator uses it
  // to identify the commands.
  // The value can be given as number literal or as string containing a binary
representation of a number. Literal
```

```
  // values must not exceed one byte in length, binary values do not have a len
gth limitation.
  "code": [
    {
      "value": 0x10
    },
    {
      "value": "000001",
      "length": 6
    }
  ],

  // The parameters of this command are used by both, the assembler and the emu
lator. They consist of a type and
  // an optional "valueNeeded"-attribute, which indicates that the value of a r
egister-parameter is not used in
  // the instructions code and should not be loaded, which can give your comman
d a serious performance boost.
  // Currently the number of parameters is restricted to 0, 1 or 2 for performa
nce reasons.
  // The total number of bits must be divisible by 8 (no half bytes).
  "params": [
    {
      "valueNeeded": false, // since this is ADD, this should be true.. but dem
o, hey!
      "type": "r_byte"
    },
    {
      // valueNeeded defaults to true here
      "type": "r_byte"
    }
  ],

  // the optional checkFlags-attribute can be used to set the zero-flag automat
ically.
  // if this is true the emulator checks the value of the first parameter for z
ero and
  // sets the flag accordingly.
  "checkFlags": { "z": true },

  // function which actually executes the command
  // the parameters are of the same types as given in the params-array.
  "exec": function (rbyte1, rbyte2) {
    var newVal = rbyte1.value + rbyte2.value;
    this.writeByteRegister(rbyte1.address, newVal & 0xFF);
    this.writeFlags((newVal > 0xFF), null);
  }
}
```

Parameter-Types

name	bitcode-length	description	value	address
r_byte	5 bits	byte-register	register-value	register-number
l_byte	8 bits	literal byte	value	null
r_word	5 bits	word-register	register-value	register-number
l_word	16 bits	literal word	value	null
pin	5 bits	pin-number	null	pin-number
address	11 bits	memory adress, label in assembler	address	null

exec-function

All parameters which are given in the `params`-array will be interpreted by the emulator and then passed on as objects into the `exec`-function. **Do not modify these objects.** They are reused for performance reasons.

```
{
  type: "",     // e.g. "r_byte"
  value: null,  // see table
  address: null // see table
}
```

There is a number of functions you can use inside the exec-function to interact with the emulator. They are documented in the class documenation. All functions marked as private must not be called.

Boilerplate

```
{
  "name": "",
  "doc": {
    "en": {
      "description": "",
      "flags": {
        "z": "",
        "c": ""
      }
    },
    "de": {
      "description": "",
      "flags": {
        "z": "",
        "c": ""
      }
    }
  },
  "tests": [
    {
      cmd: "",
      setup: { },
      steps: { }
    }
  ],
  "code": [
    {
      "value": 0x00
    }
  ],
  "params": [
    {
      "valueNeeded": true,
      "type": "r_byte"
    }
  ],
  "checkFlags": { "z": false },
  "exec": function (rbyte) {
  }
}
```

Hardware

All hardware elements in yasp share a common interface, which allows the breadboard to set and get their state but also set parameters like their color. A hardware element should always fill up all space given in its wrapper-element. Additionally it must not communicate with the emulator, but rely on the breadboard to do this. It should also never be initialized by itself, but defined in a breadboard file.
For more information about breadboards take a look at breadboards.md.
Hardware-implementations can be found in the `/src/app/js/hardware/`-directory.

Types

Currently implemented hardware types are:

- `LED`
- `PUSHBUTTON`
- `POTI`

They are documented in the /doc/hardware/-directory.

Boilerplate

```
if (typeof yasp == 'undefined') yasp = { };
if (yasp.HardwareType === undefined) yasp.HardwareType = { };

(function () {
  /**
   * ... description ...
   * Params: { }
   */
  yasp.HardwareType["PUSHBUTTON"] = {
    render: function() {
      if (!this.element) {
        // the hardware must set the this.element property
        this.element = document.createElement('div');

        // change the state, probably because the user interacted with the UI
        this.receiveStateChange(1);
      }

      // render the hardware here..
    },
    initialState: function() {
      return 0;
    }
  };
})();
```

FileDialog

This module is responsible of saving, loading, managing and creating files additionally it features a powerful driver system, allowing multiple backends.

Usage

- `yasp.FileDialog.show(mode)` shows the FileDialog, mode describing in which mode the FileDialog should be started.
  Possible parameters are:
  - `yasp.FileDialogMode.OPEN` (FileDialog in open mode)
  - `yasp.FileDialogMode.SAVE` (in save mode)
  - `yasp.FileDialogMode.SAVEAS` (in save as mode)
  - `yasp.FileDialogMode.NEW` (in new mode)
- `yasp.FileDialog.close` closes current FileDialog

Driver

Currently there are two possible ways of storing files:

- Local: This saves everything serialized as a JSON object in the localStorage
- PHP: This saves to the PHP server specifically programmed for yasp

The drivers are all callback based, so it is easily possible to transform the operations into REST requests

Structure

A Driver has to support following operations:

```
DRIVERNAME: {
  newFile: function(name, cb) {
    // newfile => cb(instance of the createdFile)
  },
  requestList: function(cb) {
    // requests list => cb(array of all the files)
  },
  deleteFile: function(name, cb) {
    // deletes file => cb(instance of the deletedFile)
  },
  renameFile: function(oldname, newname, cb) {
    // renames file => cb(successful?)
  },
  openFile: function(name, cb) {
    // opens file => cb(instance of the opened file)
  },
  saveFile: function(file, cb) {
    // saves file => cb(instance of the saved file)
  }
}
```

They should be declared in the `yasp.FileDialog.FileSystemDriver` object.

File

The files are a object that represent a specific file in the JS environment. Following attributes define a file:

```
{
  filename: name_of_file,
  username: name_of_user,
  createdate: time_of_creation,
  group: group_of_user,
  changedate: last_change_of_file,
  content: content_of_file
}
```

Files have to be serializable by `JSON.stringify`, otherwise the backend drivers do not work properly.

Communicator

Operations that take long such as assembling or do not even have an end like emulating are executed inside webworkers. On the browser-level webworkers have a simple string-messaging-api which is extended by *yasp*'s `Communicator`-class.

Specific communicator-implementations

- Assembler, see assembler.md
- Emulator, see emulator.md

Messages

Internal format of communicator-messages. The only values exposed via the API are `error`, `payload` and `action`. `id` is used only internally to coordinate the callbacks.

Normal messages

```
{
  "action": "",
  "id": 0,
  "error": 0,
  "payload": {}
}
```

Broadcasts

Broadcasts are the same as the normal messages, but their `id` will always be `null`.

API

Workers POV

```
if (typeof yasp == 'undefined') yasp = { };
importScripts(/* relative urls to scripts.. */);

var communicator = new yasp.CommunicatorBackend(self, function(data, ready) {
  // callback is called if a message is received
  switch (data.action) {
    case 'NAME':
      // reply to the message, will cause the callback on the main-thread to be
called
      ready({
        payload: {},
        error: null
      });
      break;
    default:
      // complain if an unknown action was sent
      ready(yasp.Communicator.UNKNOWN_ACTION);
      break;
  }
});
```

```
// send a broadcast
communicator.broadcast('TEST', {
  payload: {
    reason: reason
  },
  error: null
});
```

Main-Threads POV

```
var communicator = new yasp.Communicator("./url/to/file.js");

communicator.sendMessage("MSG_NAME", {
  // params of message, see documentation of the specific communicator-impl you
're working with
}, function(data) {
  var error = data.error;
  var payload = data.payload;
});

function broadHandler (data) {
  var error = data.error;
  var payload = data.payload;
}

// subscribe to an broadcast-message
communicator.subscribe("BROAD_NAME", broadHandler);

// unsubscribe again
communicator.subscribe("BROAD_NAME", broadHandler);

// kill webworker
communicator.terminate();
```

Breadboards

A breadboard is responsible for handling the hardware-elements. It wires up the hardware to the emulator, relays events and aligns the hardware as specified in one of the breadboard-files. Additionally it shows the user which hardware is attached to which pin and has other visual clues. It is possible to change the used hardware by defining a new breadboard in `/src/app/js/editor/breadboard/`.

Breadboard format

```
if (typeof yasp == 'undefined') yasp = { };
if (!yasp.BreadBoardTypes) yasp.BreadBoardTypes = { };

(function () {
  yasp.BreadBoardTypes.usbmaster = {

    // humanreadable name which is shown in menus
    name: {
      "de": "USB-Master",
      "en": "USB-Master"
    },

    // a short description which should not exceed two sentences
    description: {
      "de": "Breadboard, das aussieht wie der USB-Master",
      "en": "Breadboard which mimics the USB-Masters look"
    },

    // background-image to use (optional)
    image: {
      url: "./app/img/usbmaster.png",
      height: "260px",
      width: "335px"
    },

    // definition of the hardware-elements to create, add and wire
    hardware: [
      {
        // hardware type, see /doc/hardware.md
        type: "POTI",

        // parameters for the hardware, see files in /doc/hardware/
        params: {
          onColor: 'rgb(255,0,0)',
          offColor: 'rgb(60,0,0)'
        },

        // pin on the emulator to attach this hardware to
        pin: 11,
```

```
        // where to show the tooltip which contains additional information such
        // as used pin
        // possible values: left, right, top, bottom  (default = top)
        tooltip: {
          placement: "bottom"
        },

        // size and placement of this piece of hardware
        appearance: {
          top: "5px",
          left: "137px",
          height: "65px",
          width: "65px"
        }
      }
    ]
  };
})();
```

Boilerplate

```
if (typeof yasp == 'undefined') yasp = { };
if (!yasp.BreadBoardTypes) yasp.BreadBoardTypes = { };

(function () {
  yasp.BreadBoardTypes.CHANGEME = {
    name: {
      "de": "",
      "en": ""
    },
    description: {
      "de": "",
      "en": ""
    },
    image: { // optional
      url: "",
      height: "0",
      width: "0"
    },
    hardware: [
      {
        type: "",
        params: {
        },
        pin: 1,
        tooltip: { // optional
          placement: "top"
        },
        appearance: {
          top: "0",
          left: "0",
          height: "50px",
          width: "50px"
        }
      }
    ]
  };
})();
```

Assembler

The assembler of yasp is very simple and straightforward. It is case insensitive and implemented in a multipass architecture.
There are two different types of statements in the syntax:

Instruction

An instruction is a command string followed by zero, one or two commands (the assembler itself would support even more!).
There are two distinctions between instructions in the assembler itself, but are not visible to the end-user:

- Directives
- Commands

Directives are compile-time instructions to the assembler which are not directly converted into bitcode. Following directives currently exist:

- `DW` write a literal word into the bitcode (length: 2 bytes)
- `DB` write a literal byte into the bitcode (length: 1 byte)
- `DA` write a literal label-adress into the bitcode (length: 2 bytes)
- `ORG` set the current position in the bitcode (see Interrupts)
- `STRING` write a zero-terminated string into the bitcode (see Debugging)
- `DEFINE` defines a assemble-time constant which will be replaced before the actual assembling process
- `END`: quits current assembling at the current position, any further code will be ignored

Note: The data directives (`DW`, `DB`, `DA`, `STRING`) are capable of writing multiple values to the bitcode, simply by separation by a comma.
The directives are parsed in the function `yasp.Parser.prototype.parseDirective` in `assembler/parser.js`.

Commands are defined in the `instructions.js` file and are dynamically loaded. The commands are parsed in the function `yasp.Parser.prototype.parseCommand` in `assembler/parser.js`
For more information see Instructions.

Example

```
MOV b0, b1
```

Label

A label is a string followed by ":", indicating that it is a label. Labels are parsed in the function `yasp.Parser.prototype.parseLabel` in `assembler/parser.js`.
Important note about label names:

- Cannot have the same name as a register or similar names (b1, or b1000)
- Cannot only contain alphanumeric characters, "_" and "-"
- Cannot have the same name as an instruction or directive
- Just numbers are also not allowed

Example

```
myAwesomeLabel:
```

Multi-Pass Architecture

The assembler is divided into 4 passes:

- Lexer
- Analyser
- Parser
- Generator

The assembler is managing all the passes and holding information necessary for the editor, debugger and assembler itself. Additionally if features some helper functions for iterating over the token array (`yasp.TokenIterator`) and other utility functions (`yasp.Assembler.prototype.riseSyntaxError`).
The function `yasp.Assembler.prototype.assemble` is responsible for all the assembling. The function is returning the following data structure if assembling was successful:

```
{
  success: true,
  bitcode: new Uint8Array(),
  symbols: symbols,
  map: { }, // key: line number, value: instruction position
  ast: astArray
};
```

If compiling fails:

```
{
  success: false,
  errors: errors,
  symbols: symbols // only if it could be generated
  ast: astArray // only if it could be generated
};
```

error structure:

```
[ {
  type: errorType, // currently: minor and error, error causing the compilation to be s
topped and minor to be continued
  name: "E_ERR",
  line: error_line,
  char: error_character,
  message: error_msg
}, ...]
```

symbol structure:

```
{
  labels: { }, // key: labelname, value: label token (containing the line and character
)
  usedRegisters: { }, // key: register, value: number of times used
  defines: { }, // key: define name value: replace with
  instructions: { } // key: instruction name, value: number of times used
}
```

Lexer
This pass divides the source code into small *atomic* pieces, it also removes all comments in the input file. The result of the Lexer is an array containing all the tokens.
For more information see `assembler/lexer.js`.

Analyser
In this pass all label names and `DEFINE`s are identified. This has to be implemented so backwards definitions of labels are possible (`JMP lbl \n lbl:`), same reason applies to defines.
For more information see `assembler/analyser.js`.

Parser
The parser is responsible for doing the following tasks:
- Generating the abstract syntax "tree" (in assembler the tree is more or less just an array)
- Generating the symbol table, used by the editor and debugger
- Check semantic and syntax

For more information see `assembler/parser.js`.

Generator
The generator generates the bitcode out of the given AST, which can be executed by the emulator.
It is divided into the following sub-steps:
- Calculate bit-sizes: This one is used to get proper label addresses
- Generate bit-code: This one finally generates the bitcode.

Additionally it has following tasks:
- Generate bitcode
- Generate map (Used for converting line numbers to instruction positions)

The generator uses the `yasp.BitWriter`, which is used to write simple bit data (internally writes into a string and returns an Uint8Array).
For more information see `assembler/generator.js`.

EBNF
To give a rough overview of the syntax here is an EBNF of the syntax.

```
byteregister ::= "b0" | "b1" | "b2" | "b3" | "b4" | "b5" | "b6" | "b7" | "b8" | "b9" |
"b10" | "b11" | "b12" | "b13" | "b14" | "b15" | "b16" | "b17" | "b18" | "b19" | "b20" |
"b21" | "b22" | "b23" | "b24" | "b25" | "b26" | "b27" | "b28" | "b29" | "b30" | "b31";

wordregister ::= "w0" | "w1" | "w2" | "w3" | "w4" | "w5" | "w6" | "w7" | "w8" | "w9" |
"w10" | "w11" | "w12" | "w13" | "w14" | "w15" | "w16" | "w17" | "w18" | "w19" | "w20" |
"w21" | "w22" | "w23" | "w24" | "w25" | "w26" | "w27" | "w28" | "w29" | "w30" | "w31";

simpleliteral ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

literal ::= simpleliteral | ("0x" , simpleliteral);

symbol ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
| "n" | "o" | "p" | "q" | "r" | "s" | "t"  | "u" | "v" | "w" | "x" | "y" | "z"  |
literal;

whitespaces ::= " " , {" "};
```

```
newline ::= "\n";


labelsymbol ::= symbol | "_" | "-";

label ::= labelsymbol , { labelsymbol };

labeldef ::= label , ":" , newline;

instruction ::= symbol , { symbol };

param ::= byteregister | wordregister | literal | label;

instructiondef ::= instruction , (newline | param | (param , whitespaces , "," ,
whitespaces , param);

stringdef ::= ("string" , whitespaces , "\"" , ALL_ASCII_CHARACTERS , "\"" ,
[whitespaces, "," , "\"" , ALL_ASCII_CHARACTERS , "\""]);

datadef ::= (("db" | "dw") , whitespaces , literal , [whitespaces, "," , literal]);

dadef ::= ("da" , whitespaces , literal , [whitespaces, "," , literal]);

orgdef ::= "org", whitespaces, literal;

enddef ::= "end";

definedef ::= "define", whitespaces, label, whitespaces, label;

directivedef ::= datadef | stringdef | dadef | orgdef | enddef | definedef;

statement ::= {" "} , (labeldef | instructiondef | directivedef) , {" "} ,
newline;

program ::= { statement };
```

Testing

Commands

Testing the individual commands such as `MOV` is handled by the EmulatorTester. This class has two basic methods: `addTest` and `done`, which should be called in this order. The only argument of `addTest` is the test-case, which looks like this:

```
{
    cmd: "",
    setup: { },
    steps: [ { } ]
}
```

All three of these attributes are optional. A single step can also be written as `steps: { }`.

Process

The EmulatorTester then takes these values and does the following: 1. create an Emulator and Assembler 2. assemble the given `cmd` 3. set the `PC` to `0` 4. set all the values given in `setup` 5. load the bitcode from assembler into the emulator 4. step exactly the number of steps given, do each time:

1. execute one step in the emulator
2. check all values that should be checked
3. execute the setup given in `ss`

`setup`

- `ram`
  - `Uint8Array` replace the entire RAM
  - `object` replace individual bytes in the RAM like so: `{ 10: 42 }` writes `42` to byte `10`
- `rom` see `ram`
- `stack` see `ram`
- `reg` (`object`) change the value of an register: `{ b0: 10, pc: 3 }` possible registers:
  - `b0` - `b31` byte registers
  - `w0` - `w31` word registers
  - `pc` programm counter
  - `sp` stack pointer
- `pin` (`object`) set the state of an input or output pin: `{ 3: 1 }`
- `interruptMask` (`number`) set the interrupt-mask, see Tutorial
- `triggerInterrupt` (`number`)
- `flag` (`object`) set the carry or zero flag: `{ c: true }`
- `breakpoints` array of breakpoints, see Emulator Documentation

`steps`

Step is an array of step-objects. If there is just one step the array can be omitted: `steps: { }`. A step executes a number of checks and the issues QUnit-asserts to make sure commands work as intended. All valid steps:

- `ram` check individual bytes of the RAM: `{ 10: 42 }` (offset `10` is `42`)
- `rom` see `ram`
- `stack` see `ram`

- `reg` see `setup.reg`
- `pin` see `setup.pin`
- `interruptMask` see `setup.interruptMask`
- `flags` see `setup.flags`
- `waitTime` (`number`) check `DELAY`-instructions
- `running` (`boolean`) check if the emulator is running (`true`) or not (`false`).
- `debug` check if a debug message has been issued, see Emulator Documentation
- `ss` (`object`) execute another setup. This is done **after** all the checks.

Format of literal values

Values which are allowed for all literals, like `42` in `{ reg: { b0: 42 } }`:

- `hex` - `0xFF` - normal javascript number literal
- `dec` - `255` - normal javascript number literal
- `bin` - `"1111111"` - binary format, written as string, spaces are allowed

Full example

```
var tester = new yasp.test.EmulatorTester("demonstration");

tester.addTest({
    cmd: "ADD b0, 1",
    setup: { reg: { "b0": 0xFF } },
    steps: { reg: { "b0": 0 }, flags: { c: true, z: true } }
});

tester.addTest({
  cmd: "INV b0",
  setup: { reg: { "b0": "00000001" } },
  steps: { reg: { "b0": "11111110" }, flags: { c: false, z: false } }
});

tester.done();
```

Hardware

LED

Like a light bulb, just newer.

State

- Gets: float from `0` to `1` indicating the light-level.
- Sets: nothing

Parameters

```
{
  onColor: 'rgb(0,255,0)',
  offColor: 'rgb(0,60,0)'
}
```

PushButton

Button which sets its output to `1` as long as the user presses it. If the button is pressed while the shift-key is pressed the button will stay pressed even if the button is not clicked anymore.

State

- Gets: nothing
- Sets: `1` if pressed, else `0`

Parameters

```
{
  color: 'rgb(100,100,100)',
  pushcolor: 'rgb(60,60,60)'
}
```

Potentiometer (aka Poti)

Sets an ADC-Input-Pin to a value from `0` to `255` depending on the user input.

State

- Gets: nothing
- Sets: value from `0` to `255`

Parameters

```
{
}
```

Emulator

The emulator executes the bitcode generated by the assembler.

Ticks

The emulator works in ticks. Each tick() executes one instruction, which is fetched from the rom. Ticks executed by the tickWrapper(), which is invoked by a simple `setTimeout`. Because there is a limit on how frequent `setTimeout` can fire, a number of ticks is executed in each `tickWrapper()`-call.

1. `tickWrapper`()
    a. check if the emulator is running, if not, exit
    b. check if the number of instructions to be executed has been executed, if yes, exit (see CONTINUE-Message)
    c. check if the emulator should be waiting (`PAUSE` or `DELAY` executed), if yes, set a timeout of the desired time and exit. This is skipped if the emulator is stepping.
    d. `tick`()
        i. check for interrupts
        ii. fetch the next instruction from the ROM
        iii. check if the instruction is cached, if not disassemble and cache
        iv. fetch instruction from the cache
        v. load needed values for parameters
        vi. call the `exec()`-function of the instruction
        vii. check and set the zero-flag, if specified in instruction-file
2. `setTickWrapperTimeout`()

The timeout which is set by `setTickWrapperTimeout()` can be interrupted by a number of things:

- `CONTINUE`-message (to allow faster stepping)
- Hardware-Interrupt

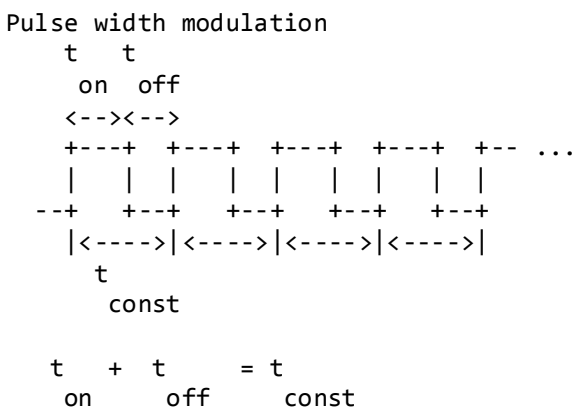
In addition the timeout will be longer when the `emulator.waitTime` (see `DELAY`-instruction) is set.

PWM

The emulator is capable of simulating PWM, in terms of calculating a pins analogue output power when a pin is rapidly switched on and off. This is done in the updatePwm() helper function. Note that this behaviour only used when the pins state changes more often than 10 times a second, otherwise the state of the pin will always be `1` or `0`. PWM causes the state to be between `1` and `255`.

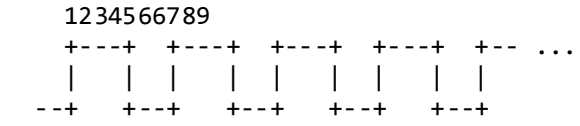
```
Impulse
    +--+
    |  |
  --+  +-----------------


Pulse
    +--+  +--+  +--+  +--+  +--+
    |  |  |  |  |  |  |  |  |  |
  --+  +--+  +--+  +--+  +--+  +--
```

Pulse width modulation



$t_{on} + t_{off} = t_{const}$

Calculation of the output power



startOn1 = 1
startOff = 5
startOn2 = 7

$t_{on} = startOff - startOn1 = 4$

$t_{off} = startOff - startOn2 = 2$

$$x = \frac{t_{on}}{t_{off} + t_{on}} = \frac{4}{6} = \frac{2}{3} = 66.66\%$$

Stack

The is located in the RAM, beginning from `0x50`. The `sp` gets increment before the byte is pushed (pre-increment), so the `sp` always points to the last byte. Words are pushed with the least significant byte first. An example can be found in the tutorial.

Data

Data structures which are used inside the emulator and its messages and broadcasts.

Messages

Details about the Communicator-Implementation for the emulator, including the messages and broadcasts.

Emulator Data

Data
All more complex data structures used by the emulator are documented here.

State
The emulator-state is used by `GET_STATE` and `SET_STATE` messages.

```
{
  "rom": new Uint8Array(),
  "ram": new Uint8Array(),
  "stack": new Uint8Array(),
  "registers": {
    "general": {
      "b": { /* indices 0-31 */ },
      "w": { /* indices 0-31 */ }
    },
    "special": {
      "pc": 0, // programm counter
      "sp": 0  // stack pointer
    },
    "flags": {
      "C": true, // carry flag
      "Z": true  // zero flag
    }
  },
  "io": [ /* IO-States */ ]
}
```

IO-State
State of an input or output pin. Used by the `IO_CHANGED`-Broadcast and `SET_STATE`-Message.

```
{
  "pin": 0,   // pin number
  "type": "", // "adc" or "gpio"
  "mode": "", // "in" or "out"
  "state": 0  // 0-255
}
```

Breakpoints

```
{
  "offset": 0,      // byte offset, or null (to break at specific statements)
  "condition": {    // can be null
    "type": "",     // "register", "flag", "io", "ram" or "rom", see table
    "param": "",    // see table
    "operator": "", // see table
    "value": null   // number, Uint8Array or null
  }
}
```

Types

type	param	value	valid operators
register	`b0 - b31` or `w0 - w31`	number (byte or word)	=, !=, <, >, <=, >=, change
flag	`c` or `z`	boolean	=, !=, change
io	pin-number	number (byte)	=, !=, change
ram	byte-offset	number (byte) or Uint8Array	=, !=, <, >, <=, >=, change
rom	byte-offset	number (byte) or Uint8Array	=, !=, <, >, <=, >=, change

Examples

stop at byte offset 8

```
{
  "offset": 8,
  "condition": null
}
```

stop at byte offset 5 if pin 3 is high

```
{
  "offset": 5,
  "condition": {
    "type": "io",
    "param": 3,
    "operator": "=",
    "value": 1
  }
}
```

stop if register **b22** is written

```
{
  "offset": null,
  "condition": {
    "type": "register",
    "param": "b22",
    "operator": "change",
    "value": null
  }
}
```

Debug

Data-structure used by the DEBUG-Broadcast.

```
{
  "type": "",
  "subtype": "",
  "addr": 0,
  "val": 0 // number or string
}
```

type	subtype	addr	val
register	b or w	register-number (0 to 31)	number
string	null	offset in ROM	string

Emulator Messages

Broadcasts

Some broadcasts (`DEBUG` and `IO_CHANGED`) are rate-limited due to the poor webworker-implementation of Firefox.

Broadcast: `BREAK`

The emulator has been stopped by a `BREAK`-Message, breakpoint, etc.

Payload

```
{
  "state": { }, // see State in doc/emulator/data.md
  "reason": ""
}
```

Reasons:

- `count`: the number of instructions to be executed has been reached, see `CONTINUE`-Message
- `break_instr`: the `BREAK`-Instruction was executed
- `divide_by_zero`: the value of the second register of a `DIV`-Instruction was zero
- `break_msg`: a `BREAK`-Message has been issued
- `invalid_instr`: a invalid instruction has been found
- `breakpoint`: a breakpoint has been hit

Broadcast: `CONTINUE`

The emulator has continued, this is caused by a `CONTINUE`-Message.

Payload

empty object

Broadcast: `LOADED`

A program has been loaded into the emulator using the `LOAD`-Message.

Payload

```
{
  start: 0,
  length: 0
}
```

Broadcast: `DEBUG`

A `DEBUG` or `ECHO`-Instruction was executed. These instructions can send the value of an register or a string stored in the ROM to the debugger. This broadcast is rate-limited to only be fired every 100 milliseconds.

Payload

```
[
  { /* debug-object, see Data-section in emulator-documentation */ }
]
```

Broadcast: `IO_CHANGED`
The state of the emulators output-pins has changed. This broadcast is rate-limited to only be fired every 50 ms for each pin, so the real number of broadcasts can be higher.

Payload
IO-State-object, see Data-section in emulator-documentation.

Messages

Message: `LOAD`
Load bitcode (probably form the assembler) into the ROM to be executed later

Message-Payload

```
{
  "bitcode": new Uint8Array(),
  "start": 0 // byte offset to start loading
}
```

Response-Payload
empty object

Error-Payloads

code	error
0	start is out of bounds
1	bitcode array too large
2	bitcode is of invalid type
3	start is of invalid type

Message: `CONTINUE`
Start or continue the execution of bitcode in the ROM. The `count`-parameter can be used to control the number of instructions executed, which is used to step through a program instead of running it. If the `skipBreakpoint`-parameter is set to `true` the execution will even be continued if the current instruction has a breakpoint set.

Message-Payload

```
{
  "count": null, // null = run unlimited steps, number = run only x steps
  "skipBreakpoint": false
}
```

Response-Payload
empty object

Error-Payloads

code	error
0	count is negative
1	count is of invalid type
2	skipBreakpoint is of invalid type

Message: BREAK
Stops the execution. This will also cause a BREAK-broadcast to be issued.

Message-Payload
empty object

Response-Payload
empty object

Error-Payloads
none

Message: GET_STATE
Get the current state of the emulator, such as registers, pc, stack, and so on.

Message-Payload
empty object

Response-Payload
state-object, see Data-section in emulator-documentation

Error-Payloads
none

Message: SET_STATE
Sets the current state of the emulator.

Message-Payload
state-object, see Data-section in emulator-documentation

Response-Payload
empty object

Error-Payloads
none

Testprogramme

Folgende Assembler Codes wurden verwendet um die Funktionalität von *yasp* sicherzustellen. Sie wurden bei der Abnahme und während der Entwicklung als Basiscodes verwendet, da sie sehr viele Aspekte des Emulators abdecken.

1. Testprogramm

```
; ====================================
;        yasp - testprogram - 01
;
; light a different LED depending on a
; potentiometers state.
;
; Values:
;     0 ...  80 => red LED
;    81 ... 165 => yellow LED
;   166 ... 255 => green LED
; ====================================

main:
        adc1    w0      ; read value from potentiometer. The value is
                        ; written to the higher byte of the given word
        debug   b1      ; output said byte

        ; test for boundary; red is between 0 and 80,
        ; so if b1 is >= 81 we must jump to yellow
        cmp     b1, 81
        jnc     m1

        ; we're below the yellow boundary, so it's red
        high    3       ; turn on red
        low     4       ; turn off yellow
        low     5       ; turn off green
        jmp     main

m1:
        ; test for boundary; yellow is between 81 and 165,
        ; so if b1 is >= 166 we must jump to green
        cmp     b1, 166
        jnc     m2

        low     3       ; turn off red
        high    4       ; turn on yellow
        low     5       ; turn off green
        jmp     main

m2:
        ; we've checked all boundaries
        ; so b1 must be between 166 and 255
        low     3       ; turn off red
        low     4       ; turn off yellow
        high    5       ; turn on green
        jmp     main

end
```

2. Testprogramm

```
; ====================================
;        yasp - testprogram - 02
;
; dimm an LED using PWM (Pulse width
; modulation). The light level of the
; LED is controllable via two buttons.
;
; Impulse
;      +--+
;      |  |
;    --+  +-----------------
;
;
; Pulse
;      +--+  +--+  +--+  +--+  +--+
;      |  |  |  |  |  |  |  |  |  |
;    --+  +--+  +--+  +--+  +--+  +--
;
; Pulse width modulation
;      t   t
;       on  off
;      <-><--->
;      +-+     +--+    +---+   +----+  +--
;      | |     |  |    |   |   |    |  |
;    --+ +-----+  +----+   +---+    +--+
;      |<----->|<----->|<----->|<----->|
;        t
;         const
;
;     t   + t    = t
;      on    off    const
;
; ====================================

        mov     w0, 1   ; T-on
        mov     w1, 200         ; T-off

main:
        ; === PWM
        high    3       ; turn the LED on
        delay   w0      ; wait for T on
        low     3       ; turn the LED off
        delay   w1      ; wait for T off

        ; === control lightlevel
        ; lighter if button 1 is pressed
        pin     1
        jnz     ligher

        ; darker if button 2 is pressed
        pin     2
```

```
        jnz     darker

        jmp main

ligher:
        inc     w0      ; T-on
        dec     w1      ; T-off
        jz      darker ; check for bounds in w1, if hit increase again
        jnz     main

darker:
        inc     w1      ; T-on
        dec     w0      ; T-off
        jz      heller ; check for bounds in w1, if hit increase again
        jmp     main

end
```

3. Testprogramm

```
; ====================================
;        yasp - testprogram - 03
;
; dimm an LED using PWM (Pulse width
; modulation). The light level of the
; LED can be changed via an potentiometer.
;
; For more information about PWM, take
; a look at testprogram 02.
; ====================================

main:
        adc1    w0      ; Read potentiometer value into the higher byte
                        ; of w0, which is b1.
                        ; This value will be used as T-on.

        mov     w1, w0 ; copy T-on to T-off
        inv     b3      ; T-on + T-off = 255

        high    3       ; turn the LED on
        delay   w0      ; wait for T-on
        low     3       ; turn the LED off
        delay   w1      ; wait for T-off

        jmp     main

end
```

4. Testprogramm

```
; ====================================
;        yasp - testprogram - 04
;
; very basic program to test various
; register commands.
; ====================================

main:
        mov     b0, 23 ; write 23 to b0
        mov     b1, 67 ; write 67 to b1
        mov     b0, b1 ; copy the content of b1 (=67) to b0
        inc     b0     ; increment b0 by 1
        add     b0, b1 ; add b0 and b1, write the result (=135) to b0
        debug   b0     ; send b0 (=135) to the debugger
        jmp     main

end
```

5. Testprogramm

```
; ====================================
;        yasp - testprogram - 05
;
; the red LED blinks and the other LEDs
; are controlled via interrupts. The
; interrupts are triggered by buttons.
; ====================================

; === init
        ; turn off all LEDs
        low     3
        low     4
        low     5

        ; turn on interrupts on pin 1 and 2
        ; Pin: 87654321
        ;  6 = 00000110b
        enable 6

; === main program
; blinks the red LED
main:
        toggle 5
        pause  5000
        jmp    main

; === interrupt service routine for pin 1
; toggles the yellow LED
isr1:
        toggle 3
        reti            ; return to normal program

; === interrupt service routine for pin 2
; toggles the green LED
isr2:
        toggle 4
        reti

; === interrupt table
; the interrupt table starts at 0x100, each entry has 2 bytes:
;   0x100 0x101 ... pin 0
;   0x102 0x103 ... pin 1
;   0x104 0x105 ... pin 2

        ; ISR for pin 1
        org    0x102
        da     isr1

        ; ISR for pin 2
        org    0x104
        da     isr2
```

6. Testprogramm

```
; ====================================
;        yasp - testprogram - 06
;
; light a LED as long as the button is
; not pressed.
; ====================================

start:
       high   3       ; turn on LED

; === loop as long as the button is not pressed
t1:
       pin    1
       jz     t1

       low    3       ; turn the LED off

; === loop as long as the button is pressed
t1n:
       pin    1
       jnz    t1n
       jmp    start

end
; ====================================
;        yasp - testprogram - 06
;
; light a LED as long as the button is
; not pressed.
; ====================================

start:
       high   3       ; turn on LED

; === loop as long as the button is not pressed
t1:
       pin    1
       jz     t1

       low    3       ; turn the LED off

; === loop as long as the button is pressed
t1n:
       pin    1
       jnz    t1n
       jmp    start

end
```

7. Testprogramm

```
; ====================================
;        yasp - testprogram - 07
;
; increment or decrement the register
; w0 by pressing two buttons. This is
; done using interrupts.
;
; For more information on interrupts,
; see testprogram 05.
; ====================================
; === init
       enable 0x06   ; enable interrupts on pin 1 and 2
       mov    w0, 500       ; initialize value
; === interrupt service routine for pin 1
main:
       toggle 3      ; toggle all LEDs
       toggle 4
       toggle 5
       debug  w0     ; send value to debugger
       jmp    main

; === interrupt service routine for pin 1
isr1:
       inc    w0
       reti
; === interrupt service routine for pin 2
isr2:
       dec    w0
       reti

; === interrupt table
       org    0x102 ; pin 1
       da     isr1
       org    0x104 ; pin 2
       da     isr2
end
```

8. Testprogramm

```
; ====================================
;        yasp - testprogram - 08
;
; toggle the red LED every .5 seconds.
; ====================================
main:
       high   3      ; turn red LED on
       delay  30000  ; wait 0.5s
       low    3      ; turn red LED off
       delay  30000  ; wait 0.5s
       jmp    main

end
```

9. Testprogramm

```
; ====================================
;        yasp - testprogram - 08
;
; Simulate a traffic light using the 3
; LEDs (red, green, yellow).
; ====================================

        MOV     W1, 60000
        MOV     W2, 20000
        MOV     W3, 40000
main:
        HIGH    3       ; red on
        DELAY   W1      ; long

        HIGH    4       ; yellow on
        DELAY   W3      ; short

        LOW     3       ; both off
        LOW     4

        HIGH    5       ; green
        DELAY   W1      ; long
        DELAY   W1
        ; loop 4 times
        ; ============
        MOV     W4, 4
blink:
        LOW     5       ; blink green LED
        DELAY   W2
        HIGH    5
        DELAY   W2

        DEC     W4
        JNZ     blink
        ; ============

        LOW     5       ; green off
        DELAY   W2

        HIGH    4
        DELAY   W3
        LOW     4

        JMP     main
```

10. Testprogramm

```
; Register:
; =========
; b0   Anzeige
; b1   Zähler
; b2   rechteste Bit beim Start
; b3   linkeste Bit beim Start

; Muster:
; =======
; 1001 0011
; 0101 0010
; 0011 0110
; 0001 1010
; 0001 1010
; 0011 0110
; 0101 0010
; 1001 0011
; 0101 0010

; fix:
; 0001 0010

; zusammensetzen:
; 0001 0010
; 0100 0010
; ---------
; 0101 0010 Rechenoperation: Oder

a:
        mov     b2, 1
        mov     b3, 128
b:
        mov     b1, 7
c:
        mov     b0, 18
        or      b0,b2
        or      b0,b3

        rl      b2
        rr      b3

        dec     b1
        jnz     c

        jmp     a
end
```

Befehlsdokumentation

ADC0 Word-Register

Deutsch: Liest einen analogen Wert von dem Pin 10 in das Register.
English: Reads an analogue value from pin 10 into the given register.

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 10010000 100$R_W R_W R_W R_W R_W$

ADC1 Word-Register

Deutsch: Liest einen analogen Wert von dem Pin 11 in das Register.
English: Reads an analogue value from pin 11 into the given register.

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 10010000 101$R_W R_W R_W R_W R_W$

ADC2 Word-Register

Deutsch: Liest einen analogen Wert von dem Pin 12 in das Register.
English: Reads an analogue value from pin 12 into the given register.

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 10010000 110$R_W R_W R_W R_W R_W$

ADC Word-Register, Pin

Deutsch: Liest einen analogen Wert von einem Pin das Register.
English: Reads an analogue value from a pin into the given register.

Anzahl der Unit-Tests: / Number of Unit-Tests: 2
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 10010000 111000$R_W R_W$ $R_W R_W R_W$PPPPP

ADD Byte-Register, Byte-Register

Deutsch: Addiert die Werte der beiden Register.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis größer als 255 (ein Byte) ist

English: Adds the values of both registers.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the result is greater than 255 (one byte)

Anzahl der Unit-Tests: / Number of Unit-Tests: 3
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 00010000
000001$R_B R_B$ $R_B R_B R_B R_B R_B R_B R_B R_B$

ADD Byte-Register, Byte-Literal

Deutsch: Addiert den Wert des Registers mit dem zweiten Parameter.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis größer als 255 (ein word) ist

English: Adds the value of the register and the given literal value.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the result is greater than 255 (one word)

Anzahl der Unit-Tests: / Number of Unit-Tests: 3
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 00000000
001$R_B R_B R_B R_B R_B$ $L_B L_B L_B L_B L_B L_B L_B L_B$

ADD Word-Register, Word-Literal

Deutsch: Addiert den Wert des Registers mit dem zweiten Parameter.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis größer als 65535 (ein word) ist

English: Adds the value of the register and the given literal value.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the result is greater than 65535 (one word)

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 00100000 001$R_WR_WR_WR_WR_W$ $L_WL_WL_WL_W$ $L_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$

ADD Word-Register, Word-Register

Deutsch: Addiert die Werte der beiden Register

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis größer als 65535 (ein word) ist

English: Adds the values of both registers.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the result is greater than 65535 (one word)

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 00010000 010001R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

AND Byte-Register, Byte-Register

Deutsch: Wendet den binären UND-Operator auf die Werte der Register an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary AND of the values of both registers.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00010000 000100R_BR_B $R_BR_BR_BR_BR_BR_BR_BR_B$

AND Byte-Register, Byte-Literal

Deutsch: Wendet den binären UND-Operator auf den Wert des Registers und den zweiten Parameter an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary AND of the value of the register and the even literal value.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00000000 100$R_BR_BR_BR_BR_B$ $L_BL_BL_BL_BL_BL_BL_BL_B$

AND Word-Register, Word-Literal

Deutsch: Wendet den binären UND-Operator auf den Wert des Registers und den zweiten Parameter an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary AND of the value of the register and the even literal value.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00100000 100$R_WR_WR_WR_WR_W$ $L_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$

AND Word-Register, Word-Register

Deutsch: Wendet den binären UND-Operator auf die Werte der Register an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary AND of the values of both registers.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00010000 010100R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

BREAK

Deutsch: Stoppt die CPU.
English: Stops the CPU.

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 00111000

CALL Label

Deutsch: Pusht die aktuelle Position als Word auf den Stack und springt dann zum gegebenen Label.
English: Pushes the current position onto the stack and jumps to the given label.

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 2

Binary Code: 11000AAA AAAAAAAA

CLR Byte-Register

Deutsch: Setzt den Wert des Registers auf 0x00.
English: Sets the value of the register to 0x00

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 2

Binary Code: 01000000 011$R_BR_BR_BR_BR_B$

CLR Word-Register

Deutsch: Setzt den Wert des Registers auf 0x0000.
English: Sets the value of the register to 0x0000

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 2

Binary Code: 01100000 011$R_WR_WR_WR_WR_W$

CMP Byte-Register, Byte-Register

Deutsch: Vergleicht zwei Werte.

- Zero-Flag wird gesetzt wenn die Werte gleich sind
- Carry-Flag wird gesetzt wenn der zweite Wert größer ist als der erste.

English: Compares two values.

- Zero-Flag is set if the values are equal
- Carry-Flag is set if the second value is greater than the first value

Anzahl der Unit-Tests: / Number of Unit-Tests: 3
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 00010000
000011R_BR_B $R_BR_BR_BR_BR_BR_BR_BR_B$

CMP Byte-Register, Byte-Literal

Deutsch: Vergleicht zwei Werte.

- Zero-Flag wird gesetzt wenn die Werte gleich sind
- Carry-Flag wird gesetzt wenn der zweite Wert größer ist als der erste.

English: Compares two values.

- Zero-Flag is set if the values are equal
- Carry-Flag is set if the second value is greater than the first value

Anzahl der Unit-Tests: / Number of Unit-Tests: 3
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 00000000
011$R_BR_BR_BR_BR_B$ $L_BL_BL_BL_BL_BL_BL_BL_B$

CMP Word-Register, Word-Literal

Deutsch: Vergleicht zwei Werte.
- Zero-Flag wird gesetzt wenn die Werte gleich sind
- Carry-Flag wird gesetzt wenn der zweite Wert größer ist als der erste.

English: Compares two values.
- Zero-Flag is set if the values are equal
- Carry-Flag is set if the second value is greater than the first value

Anzahl der Unit-Tests: / Number of Unit-Tests: **5**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 00100000 011$R_WR_WR_WR_WR_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_W$ L_WL_W

CMP Word-Register, Word-Register

Deutsch: Vergleicht zwei Werte.
- Zero-Flag wird gesetzt wenn die Werte gleich sind
- Carry-Flag wird gesetzt wenn der zweite Wert größer ist als der erste.

English: Compares two values.
- Zero-Flag is set if the values are equal
- Carry-Flag is set if the second value is greater than the first value

Anzahl der Unit-Tests: / Number of Unit-Tests: **5**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 00010000 010011R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

DEBUG Byte-Register

Deutsch: Sendet den Wert eines Byte-Registers zum Debugger.
English: Sends the value of a byte register to the debugger.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01010000 000$R_BR_BR_BR_BR_B$

DEBUG Word-Register

Deutsch: Sendet den Wert eines Word-Registers zum Debugger.
English: Sends the value of a word register to the debugger.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01110000 000$R_WR_WR_WR_WR_W$

DEC Byte-Register

Deutsch: Zieht 1 vom Wert des Registers ab. Wenn der Wert 0x00 ist, ist das Ergebnis 0xFF.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn der Wert 0 war

English: Subtracts one from the value of the register. If the value is 0x00 the result will be 0xFF.
- Zero-Flag is set if the result is 0
- Carry-Flag is set if the value was 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 01000000 001$R_BR_BR_BR_BR_B$

DEC Word-Register

Deutsch: Zieht 1 vom Wert des Registers ab. Wenn der Wert 0x00 ist, ist das Ergebnis 0xFFFF.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn der Wert 0 war

English: Subtracts one from the value of the register. If the value is 0x00 the result will be 0xFFFF.
- Zero-Flag is set if the result is 0
- Carry-Flag is set if the value was 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 01100000 001$R_WR_WR_WR_WR_W$

DISABLE

Deutsch: Deaktiviert alle interrupts
English: Disables all interrupts

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 00111010

DIV Word-Register, Word-Register

Deutsch: Dividiert den Wert des ersten Registers durch das niederwertige Byte des zweiten Registers.
English: Divides the value of the first Register with the least significant byte of the second register.

Anzahl der Unit-Tests: / Number of Unit-Tests: 4
Codezeilen / Lines Of Code (LOC): 4
Binary Code: 00010000
011000R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

ECHO, DEBUG, PRINT Label

Deutsch: Sendet einen String zum Debugger. Der String muss wie folgt definiert sind: 'stringName: STRING foobar'
English: Sends a string to the debugger. The string has to be defined like this: 'stringName: STRING foobar'

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 11010AAA AAAAAAAA

ENABLE Byte-Literal

Deutsch: Aktiviert die Interrupts, die in der gegebenen Maske binär 1 sind, z.B.: 0x03 = 00000011b = interrupts 1 und 2
English: Enables Interrupts based on the given mask, z.B.: 0x03 = 00000011b = interrupts 1 and 2

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 00000000 11100001 $L_BL_BL_BL_BL_BL_BL_BL_B$

HIGH Pin

Deutsch: Setzt den Pin auf den high- bzw. an-Zustand.
English: Puts the pin into the high- or on-state.

Anzahl der Unit-Tests: / Number of Unit-Tests: 1
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 10100000 000PPPPP

INC Byte-Register

Deutsch: Addiert 1 zum Wert des Registers. Wenn der Wert 0xFF ist, ist das Ergebnis 0x00.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn der Wert 0xFF war

English: Adds 1 to the value of the register. If the value is 0xFF, the result will be 0x00.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the value was 0xFF

Anzahl der Unit-Tests: / Number of Unit-Tests: 2
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 01000000 000$R_BR_BR_BR_BR_B$

INC Word-Register

Deutsch: Addiert 1 zum Wert des Registers. Wenn der Wert 0xFFFF ist, ist das Ergebnis 0x00.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn der Wert 0xFFFF war

English: Adds 1 to the value of the register. If the value is 0xFFFF, the result will be 0x00.
- Zero-Flag is set if the result is 0
- Carry-Flag is set if the value was 0xFFFF

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **3**
Binary Code: 01100000 000$R_W R_W R_W R_W R_W$

INV Byte-Register

Deutsch: Berechnet das Einser-Kompliment des angegebenen registers.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the 1's complement of the specified register.
- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 01000000 010$R_B R_B R_B R_B R_B$

INV Word-Register

Deutsch: Berechnet das Einser-Kompliment des angegebenen registers.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the 1's complement of the specified register.
- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 01100000 010$R_W R_W R_W R_W R_W$

JC Label

Deutsch: Springt zum gegebenen Label wenn das Carry-Flag gesetzt ist.
English: Jumps to the given label only when the carry-flag is set.

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**
Binary Code: 00001AAA AAAAAAAA

GOTO, JMP Label

Deutsch: Springt zum gegebenen Label.
English: Jumps to the given label.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**
Binary Code: 10110AAA AAAAAAAA

JMPI Word-Register

Deutsch: Springt zur adresse aus dem Word-Register.
English: Jumps to the address given in the word register.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**
Binary Code: 01110000 001$R_W R_W R_W R_W R_W$

JNC Label

Deutsch: Springt zum gegebenen Label wenn das Carry-Flag nicht gesetzt ist.
English: Jumps to the given label only when the carry-flag is not set.

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**
Binary Code: 00011AAA AAAAAAAA

JNZ Label

Deutsch: Springt zum gegebenen Label wenn das Zero-Flag gesetzt nicht ist.
English: Jumps to the given label only when the zero-flag is not set.

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**
Binary Code: 11110AAA AAAAAAAA

JZ Label

Deutsch: Springt zum gegebenen Label wenn das Zero-Flag gesetzt ist.
English: Jumps to the given label only when the zero-flag is set.

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 11100AAA AAAAAAAA

LA Word-Register, Label

Deutsch: Schreibt die Adresse eines Labels in das Register.
English: Writes the address of an label into the register.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00101000 00000000 $R_WR_WR_WR_WR_W$AAA AAAAAAAA

LOW Pin

Deutsch: Setzt den Pin auf den low- bzw. aus-Zustand.
English: Puts the pin into the low- or off-state.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 10100000 001PPPPP

MOV Byte-Register, Byte-Register

Deutsch: Kopiert den Wert des zweiten Registers in das erste.
English: Copies the value of the second register into the first one.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00010000
000000R_BR_B $R_BR_BR_BR_BR_BR_BR_BR_B$

MOV Byte-Register, Byte-Literal

Deutsch: Schreibt den gegebenen Wert in das Register.
English: Writes the given literal value into the register.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00000000
000$R_BR_BR_BR_BR_B$ $L_BL_BL_BL_BL_BL_BL_BL_B$

MOV Word-Register, Word-Register

Deutsch: Kopiert den Wert des zweiten Registers in das erste.
English: Copies the value of the second register into the first one.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00010000
010000R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

MOV Word-Register, Word-Literal

Deutsch: Schreibt den gegebenen Wert in das Register.
English: Writes the given literal value into the register.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00100000
000$R_WR_WR_WR_WR_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_WL_W$

MUL Word-Register, Word-Register

Deutsch: Multipliziert die beiden niederwertigen bytes der register.
English: Multiplies the two least significant bytes of the registers.

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00010000
010111R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

OR Byte-Register, Byte-Register

Deutsch: Wendet den binären ODER-Operator auf die Werte der Register an.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary OR of the values of both registers.
- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00010000
000101R_BR_B $R_BR_BR_BR_BR_BR_BR_BR_B$

OR Byte-Register, Byte-Literal

Deutsch: Wendet den binären ODER-Operator auf den Wert des Registers und den zweiten Parameter an.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary OR of the value of the register and the even literal value.
- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00000000
101$R_BR_BR_BR_BR_B$ $L_BL_BL_BL_BL_BL_BL_BL_B$

OR Word-Register, Word-Literal

Deutsch: Wendet den binären ODER-Operator auf den Wert des Registers und den zweiten Parameter an.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary OR of the value of the register and the even literal value.
- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00100000
101$R_WR_WR_WR_WR_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_W$ L_WL_W

OR Word-Register, Word-Register

Deutsch: Wendet den binären ODER-Operator auf die Werte der Register an.
- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary OR of the values of both registers.
- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00010000
010101R_WR_W $R_WR_WR_WR_WR_WR_WR_WR_W$

DELAY, PAUSE Word-Literal

Deutsch: Wartet die angegebene Zeit, z.B.: 1000 = 15ms, 60000 = 900ms
English: Waits a given time, e.g.: 1000 = 15ms, 60000 = 900ms

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00110000
01100000 $L_WL_WL_WL_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$

DELAY, PAUSE Word-Register

Deutsch: Wartet die im Register angegebene Zeit, z.B.: 1000 = 15ms, 60000 = 900ms
English: Waits a given time, e.g.: 1000 = 15ms, 60000 = 900ms

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01110000 010$R_WR_WR_WR_WR_W$

PIN Pin

Deutsch: Liest einen digitalen Wert von dem gegebenen Pin. low/aus z=1; high/an z=0

English: Reads a digital value from the given pin. low/off z=1; high/on z=0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 10100000 101PPPPP

POP Byte-Register

Deutsch: Nimmt das oberste Byte von dem Stack und schreibt es in das Register.
English: Takes the top byte from the stack and writes it into the register

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01000000 $111R_BR_BR_BR_BR_B$

POP Word-Register

Deutsch: Nimmt die zwei obersten Byte vom Stack und schreibt sie in das register. Das erste Byte wird zum niederwertigsten Byte des Words.
English: Takes the top two bytes from the stack and writes them into the register. The first byte becomes the least significant byte in the word.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01100000 $111R_WR_WR_WR_WR_W$

POT Pin, Word-Register

Deutsch: Liest einen analogen Wert von dem gegebenen Pin.
English: Reads an analogue value from the given pin.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00010000 100110PP $PPPR_WR_WR_WR_WR_W$

PUSH Byte-Register

Deutsch: Pusht ein byte auf den Stack.
English: Pushes one byte onto the stack.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01000000 $110R_BR_BR_BR_BR_B$

PUSH Word-Register

Deutsch: Pusht ein Word auf den Stack. Das niederwertige Byte wird zuerst gepusht.
English: Pushes one word onto the stack. The least significant byte is pushed first.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01100000 $110R_WR_WR_WR_WR_W$

RANDOM Byte-Register

Deutsch: Schreibt einen zufälligen Wert in das Register.
English: Writes a random value into the register.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 01010000 $001R_BR_BR_BR_BR_B$

RETI, RET, RETURN

Deutsch: Springt aus einer Subroutine oder einem Interrupt heraus. Es wird Word von dem Stack gelesen und als Zieladresse genutzt.
English: Jumps out of a subroutine or interrupt. One word is read from the stack and used as targetaddress.

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **1**

Binary Code: 00111011

RDROM Byte-Register, Word-Register

Deutsch: Liest den Wert im ROM an der Adresse des Word-Registers in das Byte-Register.

- Carry-Flag wird gesetzt wenn die Adresse außerhalb des RAMs liegt

English: Reads the value in ROM at the address of the word-register into the byte-register.

- Carry-Flag is set if the address is outside the bounds of the RAM

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00010000
011001R_BR_B $R_BR_BR_BR_WR_WR_WR_WR_W$

RDRAM Byte-Register, Word-Register

Deutsch: Liest den Wert im RAM an der Adresse des Word-Registers in das Byte-Register.

- Carry-Flag wird gesetzt wenn die Adresse außerhalb des RAMs liegt

English: Reads the value in RAM at the address of the word-register into the byte-register.

- Carry-Flag is set if the address is outside the bounds of the RAM

Anzahl der Unit-Tests: / Number of Unit-Tests: **1**
Codezeilen / Lines Of Code (LOC): **2**

Binary Code: 00010000
000111R_BR_B $R_BR_BR_BR_WR_WR_WR_WR_W$

RL Byte-Register

Deutsch: Shiftet das register um ein Bit nach links in das Carry-Flag. Wenn das Carry-Flag gesetzt ist, wird das rechteste Bit gesetzt. z.B.: 00100000 => 01000000

- Carry-Flag wird gesetzt wenn das höchstwertigste Bit des Wertes 1 war

English: Shifts the register by one bit to the left into the carry bit. If the carry-flag is set, the rightest bit will be set. Example: 00100000 => 01000000

- Carry-Flag is set if the most-significant bit of the value was 1

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **5**

Binary Code: 01000000 101$R_BR_BR_BR_BR_B$

RL Word-Register

Deutsch: Shiftet das register um ein Bit nach links in das Carry-Flag. Wenn das Carry-Flag gesetzt ist, wird das rechteste Bit gesetzt. z.B.: 00000010 10000000 => 00000101 00000000

- Carry-Flag wird gesetzt wenn das höchstwertigste Bit des Wertes 1 war

English: Shifts the register by one bit to the left into the carry bit. If the carry-flag is set, the rightest bit will be set. Example: 00000010 10000000 => 00000101 00000000

- Carry-Flag is set if the most-significant bit of the value was 1

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **5**

Binary Code: 01100000 101$R_WR_WR_WR_WR_W$

RR Byte-Register

Deutsch: Rotiert das Register um ein Bit nach rechts durch Carry-Flag. Wenn das Carry-Flag gesetzt ist, wird das linkste Bit gesetzt. z.B.: 00000010 => 00000001

- Carry-Flag wird gesetzt wenn das niederwertigste Bit des Wertes 1 war

English: Rotates the register by one bit to the right through the carry bit. If the carry-flag is set, the leftest bit will be set. Example: 00000010 => 00000001

- Carry-Flag is set if the least-significant bit of the value was 1

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **5**

Binary Code: 01000000 $100R_BR_BR_BR_BR_B$

RR Word-Register

Deutsch: Rotiert das Register um ein Bit nach rechts durch Carry-Flag. Wenn das Carry-Flag gesetzt ist, wird das linkste Bit gesetzt. z.B.: 00000010 10000000 => 00000001 01000000

- Carry-Flag wird gesetzt wenn das niederwertigste Bit des Wertes 1 war

English: Rotates the register by one bit to the right through the carry bit. If the carry-flag is set, the leftest bit will be set. Example: 00000010 10000000 => 00000001 01000000

- Carry-Flag is set if the least-significant bit of the value was 1

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **5**

Binary Code: 01100000 $100R_WR_WR_WR_WR_W$

SUB Byte-Register, Byte-Register

Deutsch: Subtrahiert den Wert des zweiten Registers von dem des ersten.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis kleiner als 0 ist

English: Subtracts the value of the second register from the first one.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the result smaller than 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 00010000
$000010R_BR_B$ $R_BR_BR_BR_BR_BR_BR_BR_B$

SUB Byte-Register, Byte-Literal

Deutsch: Subtrahiert den gegebenen Wert von dem Wert des Registers.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis kleiner als 0 ist

English: Subtracts the given value from the value of the first register.

- Zero-Flag is set if the result is 0
- Carry-Flag is set if the result smaller than 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **3**
Codezeilen / Lines Of Code (LOC): **3**

Binary Code: 00000000
$010R_BR_BR_BR_BR_B$ $L_BL_BL_BL_BL_BL_BL_BL_B$

SUB Word-Register, Word-Literal

Deutsch: Subtrahiert den gegebenen Wert von dem Wert des Registers.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis kleiner als 0 ist

English: Subtracts the given value from the value of the first register.

- Zero-Flag is set if the result is 0

SUB Word-Register, Word-Register

Deutsch: Subtrahiert den Wert des zweiten Registers von dem des ersten.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist
- Carry-Flag wird gesetzt wenn das Ergebnis kleiner als 0 ist

English: Subtracts the value of the second register from the first one.

- Zero-Flag is set if the result is 0

- Carry-Flag is set if the result smaller than 0

Anzahl der Unit-Tests: / Number of Unit-Tests: 3
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 00100000
$010R_WR_WR_WR_WR_W$ $L_WL_WL_WL_WL_WL_WL_WL_W$ $L_WL_WL_WL_WL_WL_W$ L_WL_W

- Carry-Flag is set if the result smaller than 0

Anzahl der Unit-Tests: / Number of Unit-Tests: 3
Codezeilen / Lines Of Code (LOC): 3

Binary Code: 00010000
$010010R_WR_W$ $R_WR_WR_WR_WR_WR_WR_WR_W$

TOGGLE Pin

Deutsch: Invertiert den digitalen Wert eines Ausgangs-Pins.
English: Inverts the digital value of an output-pin.

Anzahl der Unit-Tests: / Number of Unit-Tests: 2
Codezeilen / Lines Of Code (LOC): 1

Binary Code: 10100000 010PPPPP

WRRAM Word-Register, Byte-Register

Deutsch: Schreibt den Wert des Byte-Registers in den RAM an die adresse des Word-Registers.

- Carry-Flag wird gesetzt wenn die Adresse außerhalb des RAMs liegt

English: Writes the value of the byte-register into the RAM at the address of the word-register.

- Carry-Flag is set if the address is outside the bounds of the RAM

Anzahl der Unit-Tests: / Number of Unit-Tests: 2
Codezeilen / Lines Of Code (LOC): 2

Binary Code: 00010000
$001000R_WR_W$ $R_WR_WR_WR_BR_BR_BR_BR_B$

WRROM Word-Register, Byte-Register

Deutsch: Schreibt den Wert des Byte-Registers in den ROM an die Adresse des Word-Registers.

- Carry-Flag wird gesetzt wenn die Adresse außerhalb des RAMs liegt

English: Writes the value of the byte-register into the ROM at the address of the word-register.

- Carry-Flag is set if the address is outside the bounds of the RAM

Anzahl der Unit-Tests: / Number of Unit-Tests: 2
Codezeilen / Lines Of Code (LOC): 2

Binary Code: 00010000
$011010R_WR_W$ $R_WR_WR_WR_BR_BR_BR_BR_B$

XOR Byte-Register, Byte-Register

Deutsch: Wendet den binären Exklusiv-Oder-Operator auf die Werte der Register an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary exclusive-or of the values of both registers.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: 2
Codezeilen / Lines Of Code (LOC): 2

Binary Code: 00010000
$000110R_BR_B$ $R_BR_BR_BR_BR_BR_BR_BR_B$

XOR Byte-Register, Byte-Literal

Deutsch: Wendet den binären Exklusiv-Oder-Operator auf den Wert des Registers und den zweiten Parameter an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary exclusive-or of the value of the register and the given literal value.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00000000
$110R_BR_BR_BR_BR_B\ L_BL_BL_BL_BL_BL_BL_BL_B$

XOR Word-Register, Word-Literal

Deutsch: Wendet den binären Exklusiv-Oder-Operator auf den Wert des Registers und den zweiten Parameter an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary exclusive-or of the value of the register and the given literal value.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00100000
$110R_WR_WR_WR_WR_W\ L_WL_WL_WL_WL_WL_WL_WL_W\ L_WL_WL_WL_WL_WL_WL_W$

XOR Word-Register, Word-Register

Deutsch: Wendet den binären Exklusiv-Oder-Operator auf die Werte der Register an.

- Zero-Flag wird gesetzt wenn das Ergebnis 0 ist

English: Calculates the binary exclusive-or of the values of both registers.

- Zero-Flag is set if the result is 0

Anzahl der Unit-Tests: / Number of Unit-Tests: **2**
Codezeilen / Lines Of Code (LOC): **2**
Binary Code: 00010000
$010110R_WR_W\ R_WR_WR_WR_WR_WR_WR_WR_W$

Autorenzuordnung

	Michael Lutonsky	Robert Fischer	Dipl.-Ing. Jelinek
EINLEITUNG			
IST-Zustand	X		
mas-editor // Code-Editor	X		
mas-editor // Debugger		X	
SOLL-Zustand		X	
GRUNDLAGEN UND METHODEN			
Projektantrag	X	X	
Projektbeschreibung		X	
IST-Situation	X		
SOLL-Situation		X	
Lastenheft			X
SCRUM-Tools		X	
Risiken		X	
Technologien und Frameworks		X	
Vorgehensweise	X		
Projektmanagement	X		
Verwaltung des Source-Codes	X		
Kommunikation	X		
Automatische Tests	X		
JSDoc	X		
ERGEBNISSE			
Pflichtenheft	X	X	
Projektbeschreibung		X	
Zielbestimmung	X	X	
Soll-Ziele	X	X	
Kann-Ziele	X	X	
Nicht-Ziele	X	X	
Produkteinsatz	X		
Anwendungsbereiche		X	
Zielgruppen		X	
Betriebsbedingungen		X	
Entwicklungsumgebung	X		
Produktfunktionen und Produktumsetzung		X	
Komponentenkommunikationsübersichtsdiagramm	X		
Produktdaten		X	
Produktleistungen	X		
Anhang 1: Glossar	X	X	
Anhang 2: Befehlsdokumentation	X		
Anhang 3: Abnahmeprotokoll		X	
Anhang 5: Hardware	X		
Anhang 5: Emulatoraktivitätsdiagramme	X		
IDE		X	
Einstellungen		X	
FileManager		X	

Debugger	X		
Abnahme		X	
Veröffentlichung	X		
Abschlussbericht	X		
Anregungen für Folgeprojekte	X		
Styleguides		X	
PROJEKTDOKUMENTATION			
Zeitaufwandsauswertung	X		
Milestoneplanung		X	
Sprint Protokolle	X	X	
TECHNISCHE DOKUMENTATION			
yasp-Assembler Tutorial	X		
Localisation	X		
Instructions	X		
Hardware	X		
FileDialog		X	
Communicator	X		
Breadboards	X		
Assembler		X	
Testing	X		
Emulator	X		
Testprogramme	X		X
Autorenzuordnung		X	